Domain-Specific Programming
Announcements

▸ Tue 10/25: Guest lecture by Dr. Hongbin Zheng of AWS
  – “Systolic Array and Neuron Compiler, Key Components of a Deep Learning Accelerator”

▸ Wed 10/26 at noon in Rhodes 471: CSL seminar by Dr. Randy Huang of AWS
  – “Accelerating the Pace of AWS Inferentia Chip Development, From Concept to End Customers Use”

▸ Lab 4 due next Friday
  – Form 2-student groups on CMS today!

▸ Teaming for final project
  – Form teams on a google sheet by Thu 10/27
  – Abstract due Mon 10/31 (hard deadline)
Final Project – 35%

- In-depth exploration of a research topic
  - (1) Designing new application-specific accelerators with HLS; OR
  - (2) Devising new automation algorithms/tools
  - 3-4 students / team (11~13 teams expected in total)
    - e.g. 13 teams = 8 * 3 students + 5 * 4 students = 44 students

- Timeline
  - Project abstract due Mon 10/31
  - From Nov 1, weekly meeting with the instructor to track progress
  - Demo due Thu 12/8 (in a recorded video)
  - Final report due Wed 12/14
Agenda

▸ A brief intro to domain-specific languages (DSLs)

▸ DSLs for accelerator design
Q: Vendors of multicore processors have expressed frustration at the difficulty of moving developers to this model. As a former professor, what thoughts do you have on this transition and how to make it happen?

I might as well flame a bit about my personal unhappiness with the current trend toward multicore architecture. To me, it looks more or less like the hardware designers have run out of ideas, and that they’re trying to pass the blame for the future demise of Moore’s Law to the software writers by giving us machines that work faster only on a few key benchmarks! I won’t be surprised at all if the whole multithreading idea turns out to be a flop, worse than the "Itanium" approach that was supposed to be so terrific — until it turned out that the wished-for compilers were basically impossible to write.

void blur_filter_3x3(const Image &in, Image &blury) {
    Image blurx(in.width(), in.height()); // allocate blurx array
    for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
            blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

    for (int x = 0; x < in.width(); x++)
        for (int y = 0; y < in.height(); y++)
            blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}

The blurred face of KFC Mascot
void blur_filter_3x3(const Image &in, Image &blury) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i a, b, c, sum, avg;
        __m128i blury[(256/8)*32+2]; // allocate tile blury array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blury;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)inPtr-1);
                    b = _mm_loadu_si128((__m128i*)inPtr+1);
                    c = _mm_load_si128((__m128i*)inPtr);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurxPtr++, avg);
                    inPtr += 8;
                }
            }
            blurxPtr = blury;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i *)(&blury[yTile+y][xTile]));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurxPtr+(2*256)/8);
                    b = _mm_load_si128(blurxPtr+256/8);
                    c = _mm_load_si128(blurxPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
More for Less – Domain-Specific Languages (DSLs)

- Programming languages that are tailored for a specific application domain
  - More accessible and productive for domain experts
  - Restricted expressiveness facilitates more automated optimization and verification
  - Examples: SQL, MATLAB, OpenGL, HTML, …

- Embedded DSLs (eDSLs)
  - A DSL built on a host, typically general-purpose language
  - Examples: Halide (in C++), PyTorch (in Python), TVM (in Python), Chisel (in Scala), …
Case Study: Halide, an eDSL for Image Processing

Main Idea: Separate algorithm (what to compute) from schedule (how to compute it)

// Algorithm of Blur Filter
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

// Schedule
blurx.compute_at(blur_y, y).unroll(x);
blury.tile(x, y, xi, yi, 256, 32);

Scheduling functions encode common program transformations
- tile: loop tiling
- unroll: loop unrolling
- compute_at: change order of computation

Algorithm
- Write and test once
- Portable across platforms

Schedule
- Specify optimizations
- Explore combinations
- Target different back-ends

Modern computer systems embrace specialization to improve performance and energy efficiency

Both hardware and software are increasingly customized for dedicated application domains

Even more challenging to design and program!
Essential Techniques for Hardware Specialization

Compute customization
• parallel processing, pipelining …
Compute customization
• parallel processing, pipelining ...

Data type customization
• low-bitwidth integer, fixed point ...
Essential Techniques for Hardware Specialization

Compute customization
• parallel processing, pipelining ...

Data type customization
• low-bitwidth integer, fixed point ...

Memory customization
• banking, data reuse, streaming ...

Accelerator

Loader

PE

PE

PE

PE

PE

PE

Unloader

Scratchpad → Data movement
A Roofline View of Customization Techniques

Compute customization
• parallel processing, pipelining ...

Data type customization
• low-bitwidth integer, fixed point ...

Memory customization
• banking, data reuse, streaming ...
Building Accelerator with HLS C/C++

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] * kernel[r, c]
```

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

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

Custom compute (Loop tiling)
```
#pragma HLS pipeline II=1
```
Custom data type (Quantization)
```
ap_fixed<8,4> acc = 0;
ap_fixed<8,4> in = image[y][x];
```
Custom memory (Reuse buffers)
```
buf.shift_up(x);
buf.insert_top(in, x);
window.shift_left();
```
```
if (y >= 2 && x >= 2) {
    for (int r = 0; r < 2; r++)
        window.insert(buf.getval(r,x), i, 2);
    window.insert(in, 2, 2);
}
```
```c
out[y-2][x-2] = acc;
```
Review: 3x3 Convolution using a Line Buffer and Shift Registers

Pixels in **line buffer**
(stores 2 lines using on-chip SRAM)

1. **Push 3 pixels into shift registers** –
   1 new pixel +
   2 from line buffer

2. **Update line buffer**
   by removing the oldest pixel and shifting in the new one

**Line Buffer + Shift Registers:**
a custom “cache” + a custom “register file”
Review: Resulting Memory Hierarchy on FPGAs

- Memory architecture customized for convolution

Diagram:
- Input pixel stream
  - Flip-Flops
  - On-chip SRAMs
  - Off-chip DDR
  - Frame n
  - Frame n-1
  - Frame n-2
- Convolve
  - Output pixel stream
- Processing window
- Line buffers
- Frame buffers
HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing

Yi-Hsiang Lai\textsuperscript{1}, Yuze Chi\textsuperscript{2}, Yuwei Hu\textsuperscript{1}, Jie Wang\textsuperscript{2}, Cody Hao Yu\textsuperscript{2}, Yuan Zhou\textsuperscript{1}, Jason Cong\textsuperscript{2}, Zhiru Zhang\textsuperscript{1}

\textsuperscript{1}Cornell University, \textsuperscript{2}UCLA

\textit{FPGA’2019 (Best Paper Award)}
HeteroCL Overview

- A Python-based embedded DSL and compilation framework for productive hardware specialization
  - **Portable**: Clean decoupling of algorithm & hardware customizations
  - **Flexible**: Mixed declarative & imperative programming
  - **Efficient**: Mapping to high-performance spatial architecture templates

![HeteroCL Overview Diagram](https://github.com/cornell-zhang/heterocl)
Decoupled Compute Customization

The tensor DSL (built on TVM) separates algorithm from scheduling via declarative programming

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]))
```

Corresponding HLS code in 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]
```

Decoupled customization

```
s = hcl.create_schedule()
xo, xi = s.split(out.x, factor=M)
s.reorder(xi, xo, out.y)
```
Decoupled Data Type Customization

- HeteroCL further enables decoupled algorithm spec and data quantization schemes
  - Provides bit-accurate data type support (e.g., Int(15), Fixed(7,4))

```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]))

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

<table>
<thead>
<tr>
<th></th>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Floating-point</td>
<td>1b</td>
<td>8b</td>
<td>23b</td>
</tr>
<tr>
<td>16-bit Brain Floating-point (bfloat)</td>
<td>1b</td>
<td>8b</td>
<td>7b</td>
</tr>
<tr>
<td>8-bit Fixed-point</td>
<td>Int</td>
<td>Fraction</td>
<td>Fixed(8, 6)</td>
</tr>
<tr>
<td>2-bit Integer</td>
<td>Int</td>
<td></td>
<td>Int(2)</td>
</tr>
</tbody>
</table>

Quantize/downsize
Decoupled Memory Customization

- Inferring custom on-chip storage with `.reuse_at()`

```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 (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]

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

![Diagram showing the memory customization process]
HeteroCL further provides an embedded imperative DSL

- Not all algorithms can be described in declarative tensor-style code

```c
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]
```

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

```c
s[out].split(out.x, M)
```

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

```c
s.quantize(out, Fixed(6, 4))
```

Imperative & declarative programs share a unified interface for specifying customization primitives
## A (Partial) List of Customization Primitives in HeteroCL

<table>
<thead>
<tr>
<th>Compute customization</th>
<th>Memory customization</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.split(i, v)</code></td>
<td><code>.partition(i,v)</code></td>
</tr>
<tr>
<td></td>
<td>Partition dim i of tensor C with a factor v.</td>
</tr>
<tr>
<td><code>.fuse(i, j)</code></td>
<td><code>.reshape(i,v)</code></td>
</tr>
<tr>
<td></td>
<td>Pack dim i of tensor C into words with a factor v.</td>
</tr>
<tr>
<td><code>.reorder(i, j)</code></td>
<td><code>.buffer_at(C,i)</code></td>
</tr>
<tr>
<td></td>
<td>Create an intermediate buffer at dim i of operation C to store the results of tensor P.</td>
</tr>
<tr>
<td><code>.compute_at(C,i)</code></td>
<td><code>.reuse_at(C,i)</code></td>
</tr>
<tr>
<td></td>
<td>Create a reuse buffer storing the values of tensor P, where the values are reused at dim i of operation C.</td>
</tr>
<tr>
<td><code>.unroll(i, v)</code></td>
<td><code>.to(t, d, m)</code></td>
</tr>
<tr>
<td></td>
<td>Move a list of tensors t to destination d with mode m.</td>
</tr>
<tr>
<td><code>.parallel(i)</code></td>
<td></td>
</tr>
<tr>
<td><code>.pipeline(i, v)</code></td>
<td></td>
</tr>
</tbody>
</table>

### 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 dataflow architectures using the SODA framework. |
| C.systolic()    | Specify operation C to be implemented with systolic arrays using the AutoSA framework. |
Case Study: Matrix Multiplication (MM)

- A vanilla MM implementation performs inner product to produce one output element
  - **Floating-point accumulation** introduces carried dependency, slowing down the pipeline (II>1)

Vanilla MM using inner product approach

\[
C[i, j] = A[i, :] \cdot B[:, j]
\]
Optimized MM to Achieve II=1

- Perform a sequence of scalar-vector products to produce one output row
  - An additional buffer is required to store the intermediate results

\[ C[i,:]=\sum_{k} A[i,k] \cdot B[k,:] \]

```c
for (int i = 0; i < M; i++) {
    float C_vec[N];
    for (int j = 0; j < N; j++)
        C_vec[j] = 0.0;
    for (int k = 0; k < K; k++)
        for (int j = 0; j < N; j++)
            #pragma pipeline II=1
            C_vec[j] += A[i, k] * B[k, j];
    for (int j = 0; j < N; j++)
        C[i, j] = C_vec[j];
}
```
Optimized MM in HeteroCL

- Optimizations via decoupled primitives
  - `.buffer_at()` creates an intermediate buffer at a given axis
  - `.reorder()` swaps the order of the k and j loops
  - Algorithm code stays unchanged

```python
def MM_v2(M=1024, N=1024, K=512):
    hcl.init(hcl.Float())
    A = hcl.placeholder((M, K), name="A")
    B = hcl.placeholder((K, N), name="B")
    k = hcl.reduce_axis(0, K, name="k")
    C = hcl.compute((M, N), lambda i, j :
        hcl.sum(A[i, k] * B[k, j], axis=k), "C")

# customizations
s = hcl.create_schedule([A, B])
s.reorder(k, j)
s.buffer_at(C, i)
s.pipeline(j)
```

<table>
<thead>
<tr>
<th></th>
<th>II</th>
<th>Latency (cycles)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vanilla MM</td>
<td>8</td>
<td>4295M</td>
<td>1x</td>
</tr>
<tr>
<td>Optimized MM</td>
<td>1</td>
<td>539M</td>
<td>7.97x</td>
</tr>
</tbody>
</table>
HeteroCL Compilation Flow

```python
B = h.compute((10,), lambda x: A[k] + 2)
s = h.create_schedule()
s[B].unroll(B.axis[0])
```

Extended TVM/Halide IR

HeteroCL

General Back End

HLS Compiler
(AMD Xilinx Vivado/Vtis HLS
Intel OpenCL SDK)

AutoSA, T2SP

Spatial Architecture Templates

LLVM Code Gen

HLS Code Gen

In-memory accelerators

ASICs

FPGAs

On-going work
SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs

Yi-Hsiang Lai¹, Hongbo Rong², Size Zheng³, Weihao Zhang⁴, Xiuping Cui³, Yunshan Jia³, Jie Wang⁵, Brendan Sullivan¹, Zhiru Zhang¹, Yun Liang³, Youhui Zhang⁴, Jason Cong⁵, Nithin George², Jose Alvarez², Christopher Hughes², Pradeep Dubey²

¹Cornell University, ²Intel, ³Peking University, ⁴Tsinghua University, ⁵UCLA

ICCAD’2020
An eDSL for Constructing Systolic Arrays

Decoupled algorithm definition and spatial optimizations
Explicitly represent optimizations such as space-time transformation
Concisely describe a systolic algorithm with uniform recurrence equations (UREs)

A programming model for accelerating systolic algorithms

SuSy

Algorithm Definition
(with UREs)

Spatial Optimizations
• Space-Time Transformation
• I/O Isolation
• Vectorization ...

Processors + Accelerators

CPUs

FPGAs
Algorithm Specification with UREs

- Any systolic algorithm can be described by a set of UREs
  - i.e., an n-dimensional loop nest where the recurrences (inter-iteration dependences) must have constant distances

\[
C = A \times B
\]

for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    C[i, j] = 0;
for (int k = 0; k < N; k++)
  C[i, j] += A[i, k] * B[k, j]

Matrix Matrix Multiplication (MM) in UREs

\[
Z[i, j, k] = 0, \text{when } k = 0
\]
\[
Z[i, j, k] = Z[i, j, k - 1] + A[i, k] \cdot B[k, j], \text{when } k > 0
\]
\[
C[i, j] = Z[i, j, N - 1]
\]

Algorithm Definition in SuSy

```
// Iteration space
Var i, j, k;
// UREs
Z(i, j, k) = select(k==0, 0, Z(i, j, k-1)) + A(i, j, k) \times B(i, j, k);
C(i, j) = select(k == N-1, Z(i, j, k));
```
Space-Time Transformation

\[ T = \begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
1 & 1 & 1 \\
\end{pmatrix} \]

\[ \Pi \quad \text{Space dimensions} \quad \Pi \times (i, j, k)^T = (i, j) \]

\[ \tau \quad \text{Time schedule} \quad \tau \times (i, j, k)^T = i + j + k \]

Transformation Matrix
### Supported Spatial Optimizations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>F.merge_ures(U_1, U_2, ..., U_n)</code></td>
<td>Define the set of UREs F, U_1, U_2, ..., U_n to optimize.</td>
</tr>
<tr>
<td><code>F.space_time_transform(space, tau)</code></td>
<td>Specify the space-time transformation that will be applied to F, where space is the set of space loops, and tau is the scheduling vector.</td>
</tr>
<tr>
<td><code>F.vectorize(var)</code></td>
<td>Vectorize the specified loop variable var of F.</td>
</tr>
<tr>
<td><code>F.reorder(var_1, var_2, ..., var_n)</code></td>
<td>Reorder the loop nest for F according to the specified order, starting from the innermost level.</td>
</tr>
<tr>
<td><code>F.isolate_producer({E_1, E_2, ...}, P)</code></td>
<td>Isolate a list of expressions {E_1, E_2, ...} (usually inputs) in F to a separate producer kernel P.</td>
</tr>
<tr>
<td><code>F.isolate_consumer(E, C)</code></td>
<td>Isolate an expression E (usually an output) in F to a separate consumer kernel C.</td>
</tr>
<tr>
<td><code>F.remove(var)</code></td>
<td>Remove loop var of F.</td>
</tr>
<tr>
<td><code>F.buffer(E, v, mode)</code></td>
<td>Insert a reuse buffer at loop v for expression E with mode (either Buffer::Single or Buffer::Double).</td>
</tr>
<tr>
<td><code>F.scatter(E, var)</code></td>
<td>Reduce data communication overhead (i.e., data broadcast) by scattering the expression E to the consumer along loop var.</td>
</tr>
<tr>
<td><code>F.gather(E, var)</code></td>
<td>Reduce data communication overhead (i.e., data broadcast) by gathering the expression E from the producer along loop var.</td>
</tr>
</tbody>
</table>

Optimizations for custom I/O

https://github.com/IntelLabs/t2sp
Acknowledgements

- This lecture contains/adapts materials developed by
  - Yi-Hsiang Lai (Cornell ECE PhD, now AWS AI)
  - Authors of the following papers
    - SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs (ICCAD’20)