ECE 5775 Student-Led Discussions (10/25)

- Talks: 18-min talk + 2-min Q&A
  - Christopher Graef, Shaojie Xiang, Yanghui Ou, Peitian Pan
    Prefix Sum and Histogram
  - Neil Adit, Darshan Kumar S Yaradoni, Chirag Wadhwani, Aman Tyagi
    Serving DNNs in Real Time at Datacenter Scale with Project Brainwave
  - Yuwei Hu, Yichi Zhang, Chenhui Deng, Patrick Clobridge
    Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines

- Vote for your favorite presentation after class by Monday 10/29
  - https://goo.gl/forms/aJtTQUK0m2aoAZQr1
    - Winners receive a bonus point
  - Write down specific positive comments (minimum 80 characters)
    - Each vote is worth 0.4pt (out of 8 points allocated for student-led presentation)
  - Presenters must endorse another talk
Announcements

- Project abstract due Monday 10/29
  - From groups on CMS today!
  - Project meetings start next week

- Lab 4 due Wednesday 10/31
“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.”

Accelerator-Rich System-on-Chips (SoCs)

- Modern SoCs: hardware (HW), software (SW), customized for dedicated application domains
  - Heterogeneous processors
  - Heavily customized HW for application
  - Embedded SW tightly coupled with HW

- Embracing architectural heterogeneity and hardware specialization to improve performance and energy efficiency

Even more challenging to design & program!
Blur Filter: Original C++ Code

```c
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
Blur Filter: Optimized C++ Code for Multicore

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 blurx[(256/8)*(32+2)]; // allocate tile blurx array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurxPtr = blurx;
            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 = blurx;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (const __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);
                }
            }
        }
    }
}}}}
Domain-Specific Programming? C++ vs. Halide

// C++
for (int yo = 0; yo < Rows/TR; yo++)
for (int xo = 0; xo < Cols/TC; xo++)
for (int yi = 0; yi < Rows; yi++)
for (int xi = 0; xi < Cols; xi++)
for (int i = 0; i < N; i++)
for (int ry = 0; ry < K; ry++)
for (int rx = 0; rx < K; rx++)
out(xo*TC+xi, yo*TR+yi) += wt(rx, ry, i) *
in(xo*TC+xi+1-rx, yo*TC+yi+1-ry, i);

Optimizing C++
- Significant and intrusive code changes
- Must retest new code in case bugs are introduced
- Rewriting and retesting code for each new design point is slow

Loop Reordering
Rewrite indices
New loops

// Halide
RDom r(0,K, 0,K, 0,N);
out(x, y) += filter(r.x, r.y, r.z) * in(x+1-r.x, y+1-r.y, r.z);
out(x, y).tile(x,y, xo, xi, yo, yi, TR, TC)
    .reorder(xo, yo, xi, yi, i);
out.realize(Rows, Cols);

Optimizing Halide
- Algorithm code is unchanged
- Scheduling functions encode common optimizations
- Productive design exploration

The Halide approach\[^1\]: separate algorithm (what to compute) from schedule (how to compute it)

\[^1\] https://halide-lang.org
High-Level Programming for FPGAs

- C-based HLS entangles algorithm spec with optimization directives
  - Requires extensive code changes when targeting a different device or performance/accuracy constraint

- More recent DSLs focus on decoupling algorithm from temporal compute schedules (e.g., Halide, TVM)
Potential Project related to DSL

- HeteroCL is a DSL embedded in Python, which attempts to decouple algorithm specification from:
  - Compute customization (e.g., parallelization, pipelining)
  - Data type customization (e.g., reduced bitwidth, fixed point)
  - Memory customization (e.g., on-chip data reuse, banking)
Prefix Sum and Histogram

Christopher Graef, Shaojie Xiang, Yanghui Ou, Peitian Pan

October 25, 2018

Ryan Kastner, et al. “Chapter 8 Prefix Sum and Histogram” in Parallel Programming for FPGAs: 2018
Prefix Sum

- Want to generate sum of array up to each index, and store sum in that index
- Limited by data dependence if you do not use a buffer, so Recll is 2
- Adding a simple 1 register buffer allows for loop to achieve II of 1

```c
#define SIZE 128
void prefixsum(int in[SIZE], int out[SIZE]) {
    int A = in[0];
    for(int i=0; i < SIZE; i++) {
        #pragma HLS PIPELINE
        A = A + in[i];
        out[i] = A;
    }
}
```
Application: Counting Sort

- Counting algorithm for integer array with limited number of possible values
- Uses both prefix-sum and a histogram to order the input array
- Able to sort an array of size N with K possible values with complexity O(N+K)

```c
#define RANGE 10

void counting_sort(char in*, char out*, int size)
{
    char count[RANGE];

    // Generate histogram
    for (int i = 0; i < size; i++)
    {
        char val = in[i];
        count[val] += 1;
    }

    // Perform prefix sum on histogram
    char total = 0;
    for (int i = 0; i < RANGE; i++)
    {
        char old_count = count[i];
        count[i] = total;
        total += old_count;
    }

    // Move input values into sorted output value
    for (int i = 0; i < size; i++)
    {
        char val = in[i];
        char ind = count[val];
        output[ind] = val;
        count[val] -= 1;
    }
}
```
Application: Counting Sort

- Three Steps
  1. Create histogram of value frequencies
  2. Calculate the prefix sum of the previous histogram
  3. Store values in input array at position indexed in created histogram, decrementing each index as values are added
Histogram Computation

- Deriving a **Histogram** from the Input is similar to **Prefix summation**
- The Differences:

  **Prefix Sum**
  1. Do accumulation on output array
  2. Add input[i] each time

  **Histogram**
  1. Do accumulation for each bin
  2. Add 1 to a specific bin each time
Histogram Computation

- Deriving a **Histogram** from the Input is similar to **Prefix summation**
- The Differences:

  \[
  \begin{array}{ll}
  \text{Prefix Sum} & \text{Histogram} \\
  1. \text{ Do accumulation on output array} & 1. \text{ Do accumulation for each bin} \\
  2. \text{ Add input}[i] each time} & 2. \text{ Add 1 to a specific bin each time}
  \end{array}
  \]
Histogram Computation

- Memory read operation has latency of one cycle.
- get `hist[input[i]]`

```
for(int i = 0; i < INPUT_SIZE; i++) {
    #pragma HLS PIPELINE
    val = in[i];
    hist[val] = hist[val] + 1;
}
```

RAW: read/write the same element

1. read `val = in[i]`
2. read `hist[val]`
Histogram Computation

- Potential RAW hazard: if the elements we want to read is the same that we just wrote in last iteration

```c
for(int i = 0; i < INPUT_SIZE; i++) {
    #pragma HLS PIPELINE
    val = in[i];
    hist[val] = hist[val] + 1;
}
```

RAW: read/write the same element

1. read val = in[i]
2. read hist[val]
False Dependencies

- Since increment by 1 is cheap, we can chain it with the load or store without hurting cycle time.

- Dependency *might* occur if there exists consecutive values in the same bin.

\[ 	ext{RecMII} = \left\lceil \frac{\text{Latency}}{\text{Distance}} \right\rceil = \left\lceil \frac{2}{1} \right\rceil = 2 \]
The “dependence” Directive

- Precondition: for all $x$, $\text{in}[x] \neq \text{in}[x+1]$

- Clearly a very “dangerous” assumption

```c
void histogram(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {
    #pragma HLS DEPENDENCE variable=hist inter RAW distance=2
    int val;
    int old = -1;
    for(int i = 0; i < INPUT_SIZE; i++) {
        #pragma HLS PIPELINE
        val = in[i];
        assert(old != val);
        hist[val] = hist[val] + 1;
        old = val;
    }
}
```
Possible Solution

- Manually code some forwarding logic
  - Cost extra register and mux
  - Increase code complexity
```c
#pragma HLS DEPENDENCE variable=hist intra RAW false
for(i = 0; i < INPUT_SIZE; i++) {
    #pragma HLS PIPELINE II=1
    val = in[i];
    if(old == val) {
        acc = acc + 1;
    } else {
        hist[old] = acc;
        acc = hist[val] + 1;
    }
    old = val;
}
hist[old] = acc;
```
More depth

incr by 1 replaced by multi-cycle operations?
Improving Histogram Performance

How to achieve better performance?

Unroll the loop?

Recurrence: must complete one iteration before starting another

Partition hist[]?

No obvious access patterns
Improving Histogram Performance

Parallelize the counting phase

\[ \text{in[]} \quad 1, 4, 2, 2, 3, 1, 2, 4 \]

\[ \text{hist0[]} \quad 1, 2, 0, 1 \quad 1, 1, 1, 1 \quad \text{hist1[]} \]

\[ \text{hist[]} \quad 2, 3, 1, 2 \]

Similar to MapReduce:

Map: parallel operations on data

Reduce: reduce to the final result
Improving Histogram Performance

```c
void histogram_map(int in[INPUT_SIZE/2], int hist[VALUE_SIZE]) {
    #pragma HLS DEPENDENCE variable=hist intra RAW false
    for(int i = 0; i < VALUE_SIZE; i++) {
        hist[i] = 0;
    }
    int old = in[0];
    int acc = 0;
    for(int i = 0; i < INPUT_SIZE/2; i++) {
        #pragma HLS PIPELINE II=1
        int val = in[i];
        if(old == val) {
            acc = acc + 1;
        } else {
            hist[old] = acc;
            acc = hist[val] + 1;
        }
        old = val;
    }
    hist[old] = acc;
}

void histogram_reduce(int hist1[VALUE_SIZE], int hist2[VALUE_SIZE], int output[VALUE_SIZE]) {
    for(int i = 0; i < VALUE_SIZE; i++) {
        #pragma HLS PIPELINE II=1
        output[i] = hist1[i] + hist2[i];
    }
}
```

NUM_PE = 2

[Diagram of histogram map and reduce functions with processing elements (PE) and merge step]
void histogram(int inputA[N/2], int inputB[N/2], int hist[VSIZE]) {

    // ...

    #pragma HLS DATAFLOW
    histogram_map(inputA, hist1);
    histogram_map(inputB, hist2);
    histogram_reduce(hist1, hist2, hist);
}
Improving Histogram Performance

void histogram(int input[N], int hist[VSIZE]) {
    // parameterizable NUM_PE

    int hist_tmp[NUM_PE][VSIZE]; int input_tmp[NUM_PE][N/VSIZE+1];
    // generate input_tmp...

    #pragma HLS array_partition variable=hist_tmp complete dim=1

    #pragma HLS array_partition variable=input_tmp block factor=NUM_PE

    #pragma HLS DATAFLOW

    for (int i = 0; i < NUM_PE; i++)
    {
        histogram_map(input_tmp[i], hist_tmp[i])
    }

    histogram_reduce(hist_tmp, hist);
}
Serving DNNs in Real Time at Datacenter Scale with Project Brainwave

Neil Adit, Darshan Kumar S Yaradoni, Chirag Wadhwani, Aman Tyagi

Introduction

• Deep Neural Nets wide application base
• Microsoft web search and speech to text for Bing
• RNNs and LSTMs more memory and compute intensive than CNNs
• Standard CPU growth not sufficient
• GPUs to rescue → high throughput for offline training
• Need power efficient to deploy
• Low latency for real time cloud applications
• Enter Brainwave
Recurrent Neural Networks (RNNs)

Not effective for learning long-term dependencies due to vanishing gradients!

Image reference: http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Adds a unitary path/ storage cell for "remembering" previous inputs
Brainwave Background

- Mesh of high-end FPGAs attached to network
- 100K FPGAs communicate under 10us
- Pooling of FPGAs to share workload and rebalancing underutilized resources
Architecture Stack

• DNN in desired framework -> Graph Intermediate Response (IR) with tensor operation nodes and dataflow edges
• Graph partitioning
• Compiling using device specific tools
Brainwave Architecture

- Exploit parallelism and On-chip pinning at scale
  - Split DNN into subgraph for FPGA/CPU
  - Subgraph fits entirely on-chip allowing terabytes/sec memory bandwidth
  - If on chip memory exhausted -> scale hardware microservice
  - Stand alone NPUs either scale down models or store off chip
Graph partitioning

• Type of layer
  • CNNs have high compute intensity, low memory requirement – single FPGA
  • RNNs have large weight matrices – mapped greedily over multiple FPGAs

• Operators
  • FPGA unsupported or not profitable operators / “nodes” in IR mapped to CPU
Compiling graphs based on CPU/FPGA

- CPU executables are generated using optimized libraries such as BLAS
- For FPGAs
  - HLS: convenient but not suitable for programmable hardware
  - Hand optimized RTL: highly efficient but time consuming
  - NPUs with custom architecture and instructions built for this purpose
Brainwave NPU

Compile-time narrow precision data types
- Higher performance than what is possible using conventional float and integer types

Single-threaded model with an extensible ISA
- Adapts to fast-changing DNN algorithms

Scalable microarchitecture
- Maximizes hardware efficiency at low batch sizes

Mega-SIMD

Single issued instruction produces a million operations 130,000 ops per cycle over 10 cycles
Brainwave NPU

*Hardware Organization*

• Mainly consists of NFU – Neural Functional Unit
  • NFU consists of specialized functional units for accelerating common-case DNN operations – dense matrix multiplications or activations

• Matrix Vector Unit (MVU)
  • Largest functional unit in Brainwave NPU.
  • Attached to secondary MFUs (multifunctional units) that implement activations and element-wise vector-vector operations
Brainwave NPU

*Hardware Organization*

- MVU consists of thousands of parallel MAC (Multiply Accumulators)
  - MACs organized into parallel multi-lane dot product units
  - Maintains high utilization of all MAC resources when evaluating individual matrix-vector multiplication operations at batch size of 1
Brainwave NPU

*Hardware Organization*

- Resizing activation matrix to multiple rows increases batch size and increases hardware utilization
  - But does not improve performance

- Brainwave NPU makes use of abundant independent on-chip resources of FPGAs

| 11,721 independently addressable 512x40b SRAM | 30MB on-chip storage | 35 terabytes/sec BW at 600MHz |

Intel Stratix 10 280 FPGA
Brainwave NPU

*Narrow Precision*

- DNNs can maintain acceptable levels of accuracy even when weights and activations are expressed in low numerical precisions.
- It is one strategy to achieve a boost in compute performance as well as power efficiency.
- Neural-optimized data formats based on 8 and 9 bit floating point are developed, where mantissas are trimmed to 2 or 3 bits.
- Formats referred to as ms-fp8 and ms-fp9. It can achieve higher dynamic range and accuracy than fixed point implementations.
Narrow Precision

Use of ms-fp8 narrow precision improves performance by 3.2-7.8 times over a conventional 16 bit fixed point.
Narrow Precision

- A non-retrained conversion results in a minimal degradation in accuracy
- Can be fully recovered with 1-3 stages of quantized training
Results

Accelerating Bing Intelligent search

- Comparison of CPU-only vs. Brainwave-accelerated Turing prototype 1 (TP1) and DeepScan DNN models in Bing production.
- Brainwave accelerated models showing substantial improvement in latency.

<table>
<thead>
<tr>
<th></th>
<th>Bing TP1</th>
<th>Bing DeepScan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU-only</td>
<td>Brainwave-accelerated</td>
</tr>
<tr>
<td></td>
<td>GRU 128x200 (x2) + W2Vec</td>
<td>LSTM 500x200 (x8) + W2Vec</td>
</tr>
<tr>
<td></td>
<td>9 ms</td>
<td>0.850 ms</td>
</tr>
<tr>
<td>End-to-end latency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>per Batch 1 request</td>
<td></td>
<td></td>
</tr>
<tr>
<td>at 95%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1D CNN + W2Vec (RNNs removed)</td>
</tr>
<tr>
<td></td>
<td>15 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td></td>
<td>1D CNN + W2Vec + GRU 500x500 (x4)</td>
<td></td>
</tr>
</tbody>
</table>
Brainwave NPU on Pre-production Stratix 10 280

- **Intel Stratix 10 280 FPGA** (based on 14 nm tri gate process) has significant improvement over previous FPGA generations.

- Compared to Stratix D5 FPGA:
  - 3 times higher clock speed (>600 Mhz)
  - 3.6 times more DSPs
  - 5.4 times more soft logic resources
  - 5.8 times additional on-chip memory
Brainwave NPU on Pre-production Stratix 10 280

- Brainwave NPU running on Intel Stratix 10 280 FPGA serves a Batch-1 high-dimensional 3,200x2,400 GRU with 480 timesteps in under 1 ms.
- This measurement amounts to 39 billion operations on this benchmark which is 5 times that of ResNet-50 and about 25 times of AlexNet.
- This achieves an effective throughput of 39.5 TFLOPs out of 48 TFLOPs peak (82% efficiency) at batch 1 and energy efficiency of 720 GOPs/Watt.
Conclusion

- Project BrainWave is a powerful platform for an accelerated AI cloud.
- Runs on Microsoft’s hyperscale infrastructure with FPGAs.
- It serves state of the art, pre trained DNN models at ultra-low latency at Batch 1 without compromising on throughput and efficiency.
- Narrow Precision quantization enables the Project Brainwave system to achieve competitive levels of performance and energy efficiency.
Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines

Yuwei Hu, Yichi Zhang, Chenhui Deng, Patrick Clobridge

Image Processing Pipelines

This is used in cameras to improve image quality
Stencil Computing in Image Processing

- Compute intensive in general
- Opportunities for parallelism
- Opportunities for data reuse
Image Signal Processors (ISPs)

ISPs are specialized hardware implemented in ASIC
➢ 500x more power efficient than CPU

But, ASIC design is not easy

Solution: Domain Specific Language (DSL) -> Hardware

```
bx = im(x,y)
   (I(x-1,y) +
   I(x,y) +
   I(x+1,y))/3
end
by = im(x,y)
   (bx(x,y-1) +
   bx(x,y) +
   bx(x,y+1))/3
end
sharpened = im(x,y)
   I(x,y) + 0.1*
   (I(x,y) - by(x,y))
end
```

Stencil Language
Darkroom DSL

\[ O_1 = \text{im}(x,y) \]
\[ \text{In}(x-1,y) + \text{In}(x,y) + \text{In}(x+1,y) \]
end

\[ O_2 = \text{im}(x,y) \]
\[ O_1(x,y-1) + O_1(x,y) + O_1(x,y+1) \]
end

```
for(i=0;i<width;i++)
{
    for(j=0;j<height;j++)
    {
        Out[i][j] = 0;
        Out[i][j] += In[i][j-1];
        Out[i][j] += In[i][j];
        Out[i][j] += In[i][j+1];
    }
}
```

```
for(i=0;i<width;i++)
{
    for(j=0;j<height;j++)
    {
        Out[i][j] = 0;
        Out[i][j] += In[i-1][j];
        Out[i][j] += In[i][j];
        Out[i][j] += In[i+1][j];
    }
}
```
Advantages of DSL

1. Intuitive (clear data flow)

2. Compact

3. Easy for optimization

4. Cross-platform (run on multiple platforms)
Compiling DSL into Hardware

Stencil Language

\[
\begin{align*}
\text{bx} &= \text{im}(x,y) \\
&= (I(x-1,y) + I(x,y) + I(x+1,y))/3 \\
\text{end} \\
\text{by} &= \text{im}(x,y) \\
&= (bx(x,y-1) + bx(x,y) + bx(x,y+1))/3 \\
\text{end} \\
\text{sharpened} &= \text{im}(x,y) \\
&= I(x,y) + 0.1* (I(x,y) - by(x,y)) \\
\text{end}
\end{align*}
\]
Inserting Line Buffer

\[ \text{out}[x] = \text{in}[x] + \text{in}[x - 1] + \text{in}[x - 2] \]

Processing Function

Pipeline Model

Line Buffer

Part of the figure from Darkroom Paper (2014)
How Line Buffer Works

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>In[1]</td>
<td>In[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Part of the figure from Darkroom Paper (2014)
Non-Causal Filters

The output relies on values in the future

\[
C(x) = \frac{A(x)}{(B(x+1) + B(x))}
\]

\[
D(x) = C(x+1) + C(x)
\]

Naive schedule for this code:
Shift Operator

\[ f_s(x) = f(x - s) \quad s = 1 \]

\[ C_1(x) = \frac{A(x-1)}{B(x) + B(x-1)} \]
Shift Operator

\[\text{fs}(x) = f(x - s)\]

When \( s = 1 \):

\[C_1(x) = \frac{A(x-1)}{B(x) + B(x-1)}\]

When \( s = 2 \):

\[D_2(x) = C_1(x) + C_1(x-1)\]
Non-Causal Pipeline

We can minimize line buffer size by using ILP
Optimized Pipeline
Implementation

Front-end Optimization:
- Converts code into IR

DAG linearization (only hardware):
- Flattens the DAG into a linear pipeline (each node has only one preceding and succeeding node)

ILP:
- Generate line buffer pipeline from the IR

CPU Code Gen:
- Converts the line buffer pipeline into x86 code.

Verilog Code Gen:
- Converts the line buffer pipeline into Verilog.

ASIC toolchain:
- Converts Verilog into input to design the ASIC

FPGA toolchain:
- Converts Verilog into a bitstream for the FPGA
Results
Results
Conclusion

<table>
<thead>
<tr>
<th></th>
<th>HLS</th>
<th>Darkroom and other DSLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction level</td>
<td>high</td>
<td>higher</td>
</tr>
<tr>
<td>Programming difficulty</td>
<td>easy</td>
<td>easier</td>
</tr>
<tr>
<td>Application</td>
<td>general</td>
<td>domain specific</td>
</tr>
<tr>
<td>Output</td>
<td>Verilog</td>
<td>Verilog/HLS</td>
</tr>
</tbody>
</table>

- DSL is great for domain specific applications
- A hot research topic recently: [Halide](#), [TVM](#), [TACO](#), ...