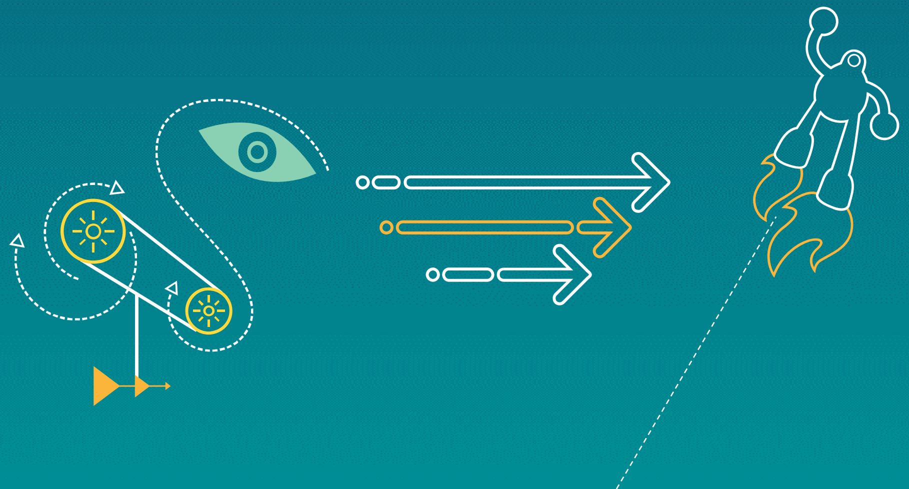
# Halide for HVX Training

Qualcomm Technologies, Inc.

80-PD002-3 Rev. A



Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

# Revision History

| Revision | Date | Description |
|---|---|---|
| A | December 2017 | Initial release |

# Course Outline

- Part 1 – Introduction to Halide
- Part 2 – Halide Programming Language
  - Algorithms
  - Schedules
- Part 3 – Halide for Qualcomm® Hexagon™ Vector eXtensions (HVX)
  - Getting Started/Requirements
  - Hexagon SDK
  - Hexagon Tools
  - Halide Tools
- Part 4 – Performance Optimizations
  - Halide for HVX modes
  - Performance
  - Profiling
  - Suggested Scheduling Techniques
- Part 5 – Installation Instructions, engineering related
  - Examples
  - Simulated Testing
  - MTP Device (offload, standalone)
  - Debugging

# Prerequisites

Basic knowledge of the following is assumed.

- Hexagon architecture with HVX
- ARM architecture and the challenges involved in heterogenous computing
- The basics of image processing for computational photography

# Part 1 – Introduction to Halide

1

**Introduction to Halide**

2

The Halide Programming Language

3

Halide for HVX

4

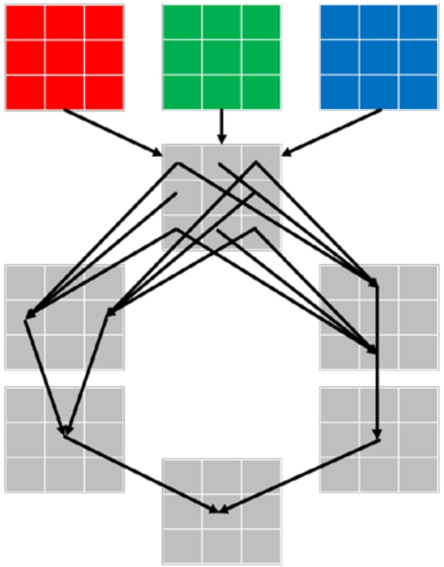
Performance Optimizations

5

Installation Instructions and Troubleshooting

# Halide

- A new domain specific language (DSL) for image processing and computational photography
- Fast image-processing pipelines are difficult to write
  - Definition of the stages of the pipeline
  - Optimization of the pipeline - vectorization, multi-threading, tiling, etc
- Traditional languages make expression of parallelism, tiling and other optimizations difficult
- Solution – Halide enables rapid authoring and evaluation of optimized pipelines by separating the algorithm from the computational organization of the different stages of the pipeline (schedule)
- Programmer defines both the algorithm and the schedule
- Front end embedded in C++



Sobel edge detection

# Halide (cont.)

- A new DSL for image processing and computational photography.
- Halide programs / pipelines consist of two major components
  - Algorithm
  - Schedule
- Algorithms specify what is computed at a pixel
- Schedules specify how  the computation is organized

```
// Image with 8 bits per pixel.
ImageParam input(UInt(8), 2);
// A pipeline stage
Halide::Func f;

// horizontal blur – Algorithm.
f(x, y) =  (input(x-1, y) + input(x, y) + input(x+1, y))/3;
// Schedule
f.vectorize(x, 128).parallel(y, 16);
```

# Typical Optimization Techniques for Image Processing

Blue boxes represent memory read and yellow represent memory written

Baseline case – Compute the producer fully before computing the consumer (poor cache locality)

**Note:** Images courtesy halide-lang.org. Click the images to open the .gif in a web browser and view the animation.

# Typical Optimization Techniques for Image Processing – Tiling

[redacted]

Organize the computation of the consumer in "tiles" and compute only the amount of the producer needed for the each tile. This ensures that the values of the producer needed for subsequent rows of the consumer are still in the cache (better cache utilization).

# Typical Optimization Techniques for Image Processing – Vectorization and Parallelization

Tile the computation of the consumer and vectorize the horizontal dimension of the tiles. Divide the rows of the consumer to be computed by 4 threads

# Scheduling for Performance – Halide vs. C++ Comparison

```
  // horizontal blur – Algorithm.
  f(x, y) =  (input(x-1, y) + input(x, y) +
               input(x+1, y))/3;
```

```
for (y = min_row; y < max_row; ++y) {
  for (x = min_col, x <= max_col; ++x) {
    f(x, y) =
          (input(x-1,y) + input(x, y) + input(x+1, y)) / 3;
   }
}
```

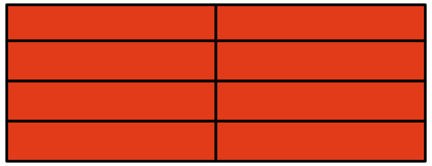
# Scheduling for Performance – Halide vs. C++ Comparison (cont.)

```
  // horizontal blur – Algorithm.
  f(x, y) =  (input(x-1, y) + input(x, y) +
              input(x+1, y))/3;
```

```
for (y = min_row; y < max_row; ++y) {
  for (x = min_col, x <= max_col; ++x) {
    f(x, y) =
          (input(x-1,y) + input(x, y) + input(x+1, y)) / 3;
   }
}
```

- Tiling

```
f.tile(x, y, xi, yi, 128, 4);
```



```
for (y = min_row; y < max_row/4; ++y) {
  for (x = min_col, x <= max_col/128; ++x) {
    for(yi = 0; yi < 4; ++yi) {
      for (xi = 0; xi < 128; ++xi) {
        x_ = x*128 + xi;
        y_ = y*4 + yi;
        f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                     input(x_+1, y_)) / 3;
      }
   }
}
```

# Scheduling for Performance – Halide vs. C++ Comparison (cont.)

- Tiling and unrolling
- Benefits of unrolling
  - Reduces branching overhead
  - Increases instruction level parallelism (ILP) which is especially advantageous on a processor like Hexagon which is a VLIW processor
  - Exposes dependences across loop iterations such as read-after-write (RAW) dependences across two iterations of a loop

```
f.tile(x, y, xi, yi, 128, 4)
 .unroll(yi);
```

```
for(y = min_row; y < max_row/4; ++y) {
  for (x = min_col, x <= max_col/128; ++x) {
    for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y*4 + 0;
      f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                   input(x_+1, y_)) / 3;
   }
    for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y*4 + 1;
      f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                   input(x_+1, y_)) / 3;
   }
   for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y*4 + 2;
      f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                   input(x_+1, y_)) / 3;
   }
   for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y*4 + 3;
      f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                   input(x_+1, y_)) / 3;
   }
}
```

# Scheduling for Performance – Halide vs. C++ Comparison (cont.)

- Tiling, unrolling, and vectorization

```
f.tile(x, y, xi, yi, 128, 4)
 .unroll(yi)
 .vectorize(xi);
```

```
for(y = min_row; y < max_row/4; ++y) {
  for (x = min_col, x <= max_col/128; ++x) {
    for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y*4 + 0;
      f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                   input(x_+1, y_)) / 3;
   }
 for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y
      f(x_,

   }
 for (xi =
      x_ =
      y_ =
      f(x_,

   }
 for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y*4 + 3;
      f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                   input(x_+1, y_)) / 3;
   }
}
```

Not possible without an auto-vectorizer

# Scheduling for Performance – Halide vs. C++ Comparison (cont.)

- Tiling, unrolling, and vectorization

```
f.tile(x, y, xi, yi, 128, 4)
 .unroll(yi)
 .vectorizer(xi);
```

Halide provides many more scheduling directives for greater control over the organization of computation. The Halide user guide and the tutorials hosted at `halide-lang.org` are the best resources for learning more about these scheduling directives

```
for(y = min_row; y < max_row/4; ++y) {
  for (x = min_col, x <= max_col/128; ++x) {
    for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_
      f

   }
 for (x
      x
      y
      f

   }
 for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y*4 + 2;
      f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                   input(x_+1, y_)) / 3;
   }
 for (xi = 0; xi < 128; ++xi) {
      x_ = x*128 + xi;
      y_ = y*4 + 3;
      f(x_, y_) = (input(x_-1,y_) + input(x_, y_) +
                   input(x_+1, y_)) / 3;
   }
}
```

Not possible without an auto-vectorizer

# Summary

- Halide programs consist of two major components
  - Algorithm
  - Schedule
- The separation between algorithm and schedule is key. It enables:
  - Authoring an algorithm without the complexity of computational organization
  - Tuning and experimenting with the computational organization to achieve high performance
- Generation of highly optimized compiled code
  - The compiler is guided by the programmer explicitly specifying high-level optimizations

# Part 2 – The Halide Programming Language

1

Introduction to Halide

2

**The Halide Programming Language**

3

Halide for HVX

4

Performance Optimizations

5

Installation Instructions and Troubleshooting

# Halide Terminology – Func, Expr, and Var

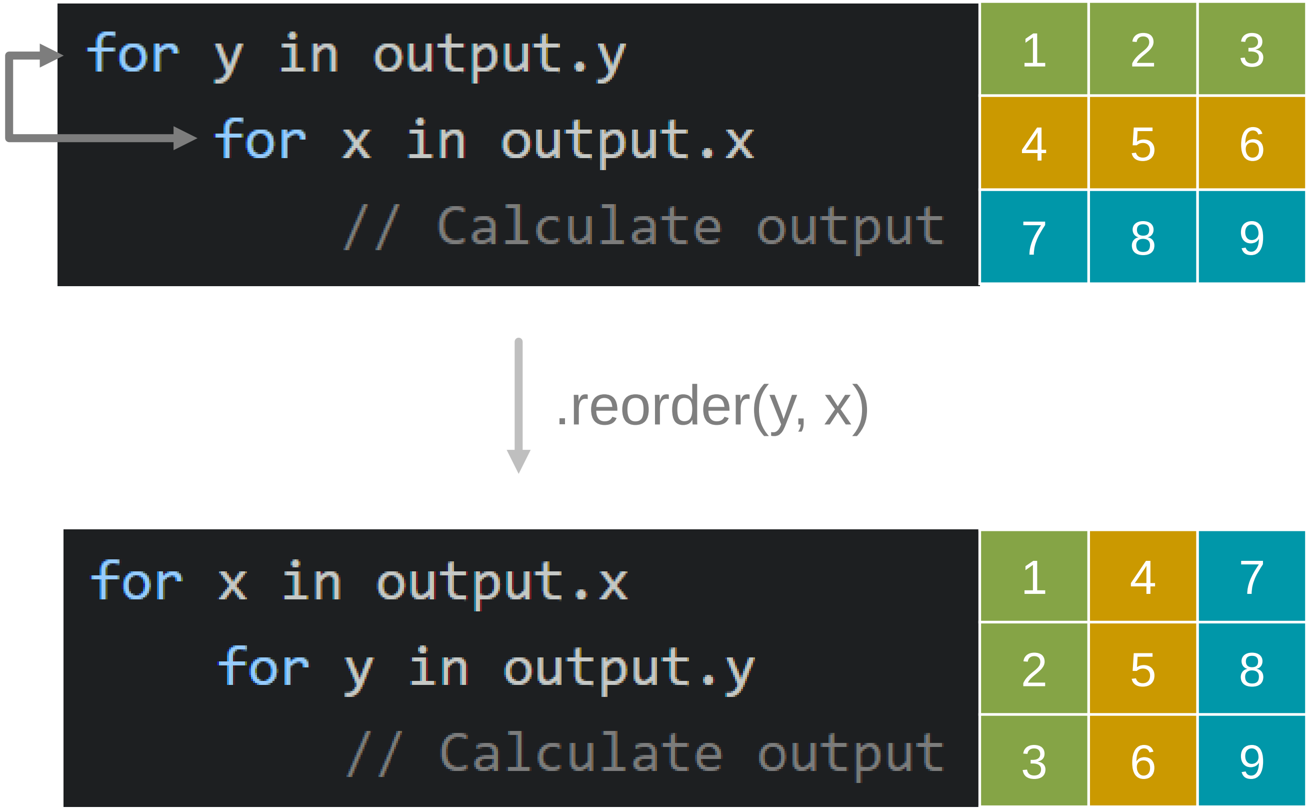
- Func:
  - Halide models image as function (f ), s.t. f : I → R
  - Bounds are implicitly calculated using input and output image sizes
- Var:
  - Used to define a Func
  - Have no meaning on their own
- Expr:
  - Composed of Funcs, Vars and other Exprs

```
Expr e = x + y;
f(x, y) = 3*e + x;
g(x, y) = min(f(x-1, y), f(x, y), f(x+1, y), e);
```

x, y → Var
e → Expr
f, g → Func

# Scheduling a Stage – Reordering Dimensions

- Change loop orders
- Order is defined from innermost to outermost loop



```
for y in output.y
    for x in output.x
        // Calculate output
```

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

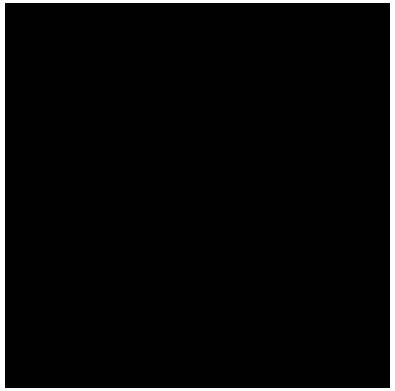
.reorder(y, x)

```
for x in output.x
    for y in output.y
        // Calculate output
```

| | | |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

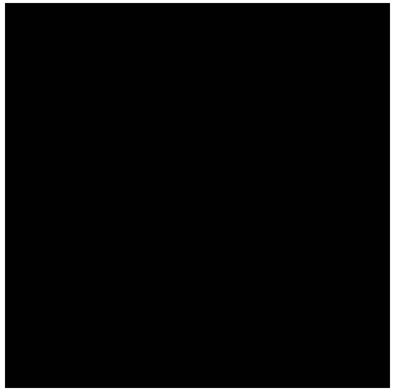
- Usage:
  - `output.reorder(dimensions from innermost > outermost)`

# Scheduling a Stage – Reordering Dimensions (cont.)



Row Major Traversal

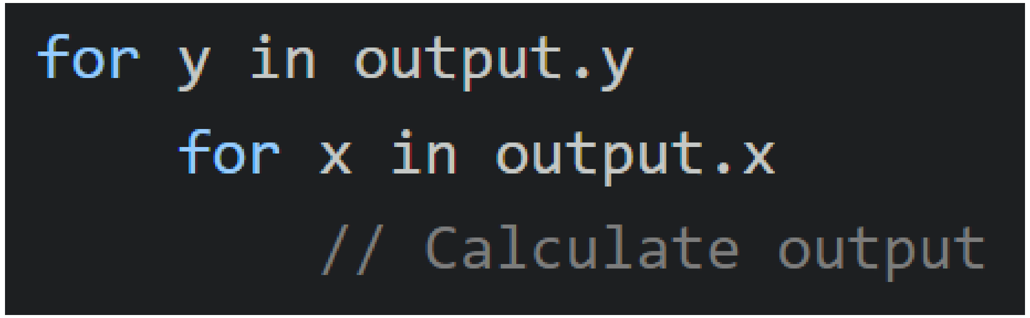
reorder(y, x)



Column Major Traversal

*Images, animations and pipeline credits: Halide-lang.org

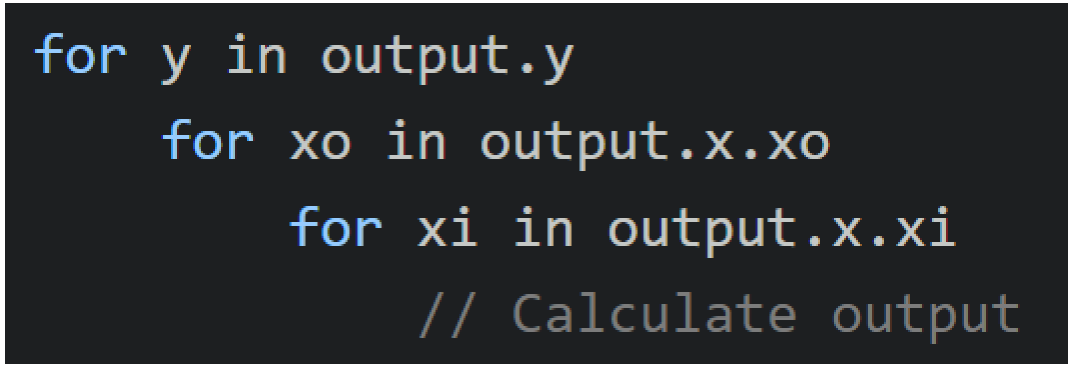
# Scheduling a Stage – Splitting a Dimension



```
for y in output.y
    for x in output.x
        // Calculate output
```

|  |  |  |  |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

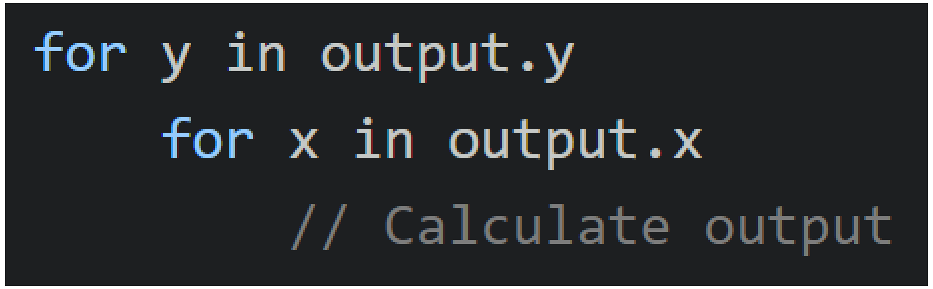
.split(x, xo, xi, split_factor)



```
for y in output.y
    for xo in output.x.xo
        for xi in output.x.xi
            // Calculate output
```

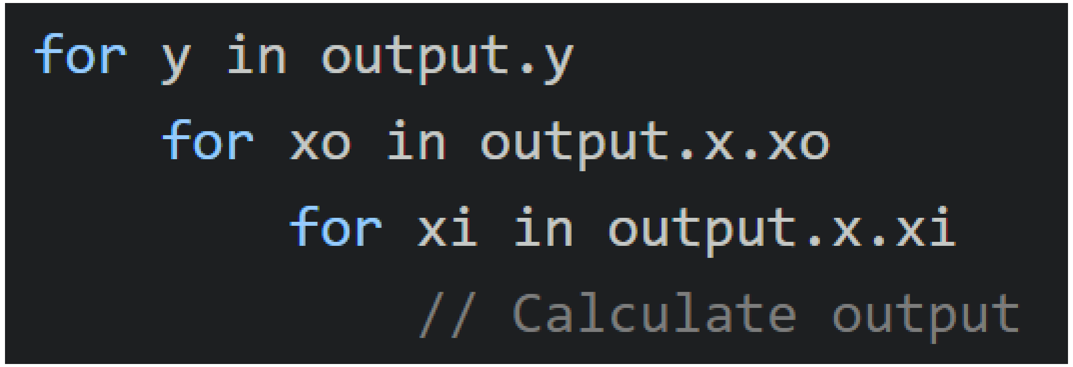
|  |  |  |  |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

# Scheduling a Stage – How Does Splitting Help



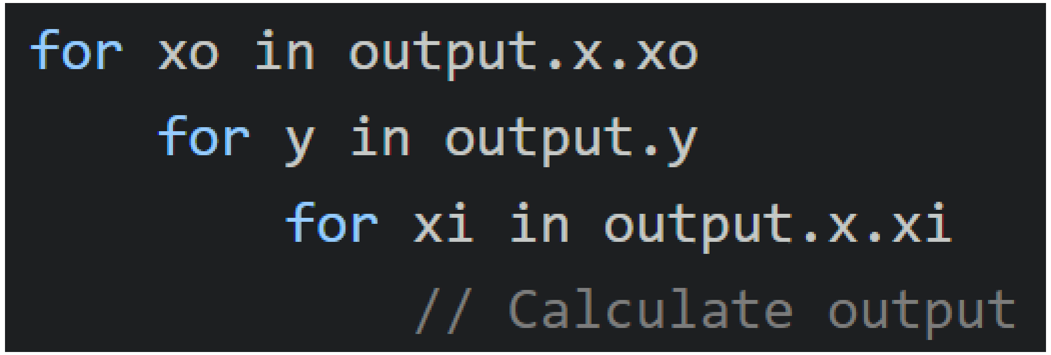
```
for y in output.y
    for x in output.x
        // Calculate output
```

.split(x, xo, xi, split_factor)



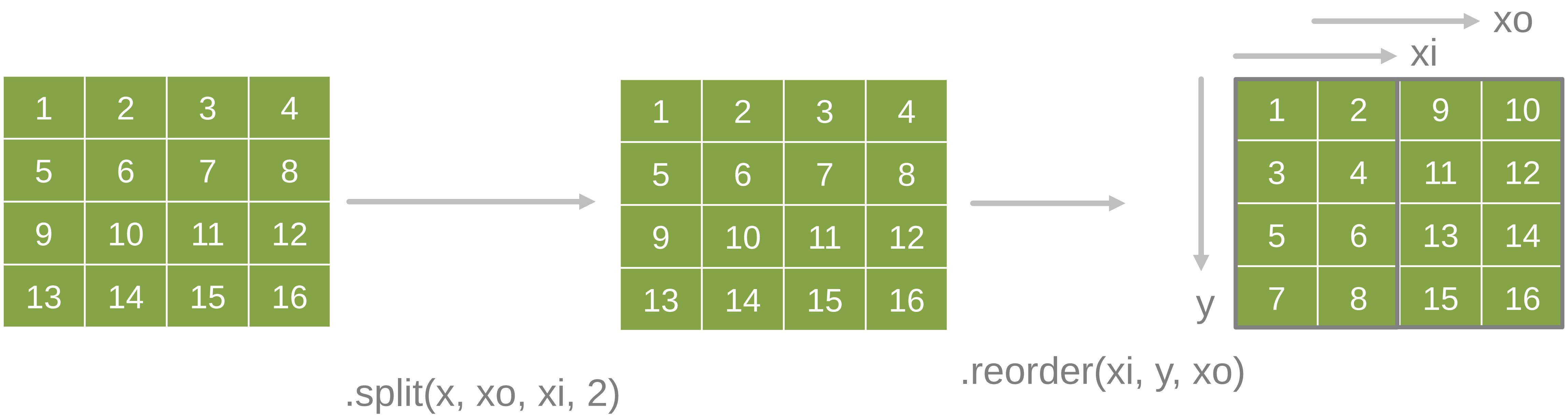
```
for y in output.y
    for xo in output.x.xo
        for xi in output.x.xi
            // Calculate output
```

.reorder(xi, y, xo)



```
for xo in output.x.xo
    for y in output.y
        for xi in output.x.xi
            // Calculate output
```

# Scheduling a Stage – How Does Splitting Help (cont.)



|  |  |  |  |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

.split(x, xo, xi, 2)

|  |  |  |  |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

.reorder(xi, y, xo)

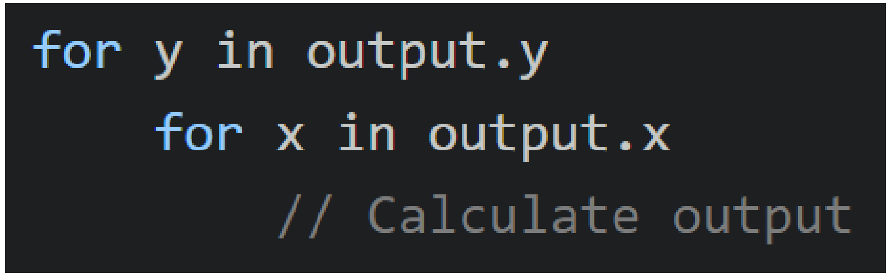
xo

xi

y

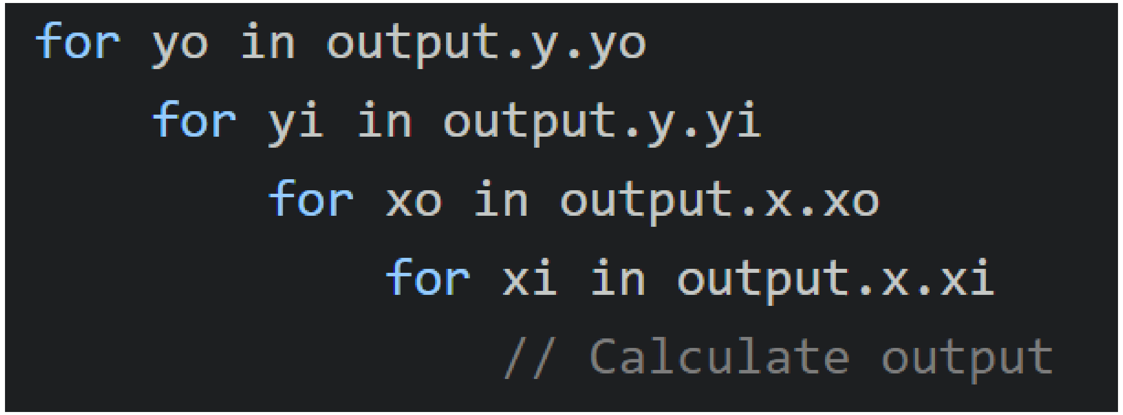
|  |  |  |  |
|---|---|---|---|
| 1 | 2 | 9 | 10 |
| 3 | 4 | 11 | 12 |
| 5 | 6 | 13 | 14 |
| 7 | 8 | 15 | 16 |

# Scheduling a Stage – Tiling



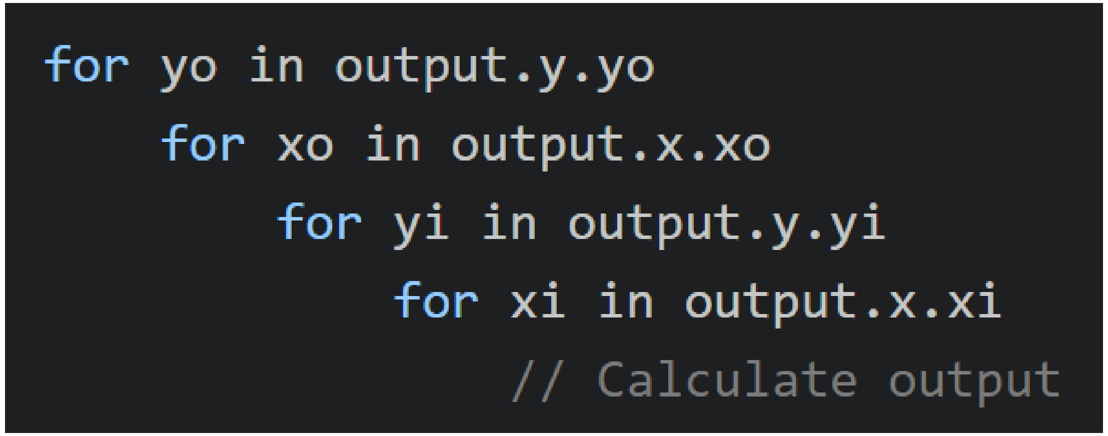
```
for y in output.y
    for x in output.x
        // Calculate output
```

.split(x, xo, xi, split_factor)
.split(y, yo, yi, split_factor)



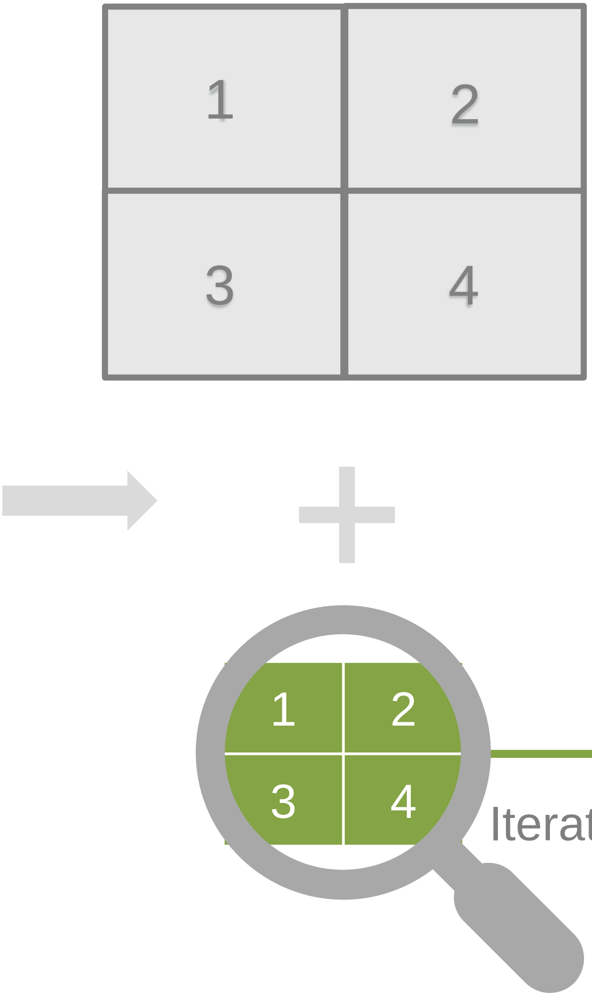
```
for yo in output.y.yo
    for yi in output.y.yi
        for xo in output.x.xo
            for xi in output.x.xi
                // Calculate output
```

.reorder(xi, yi, xo, yo)

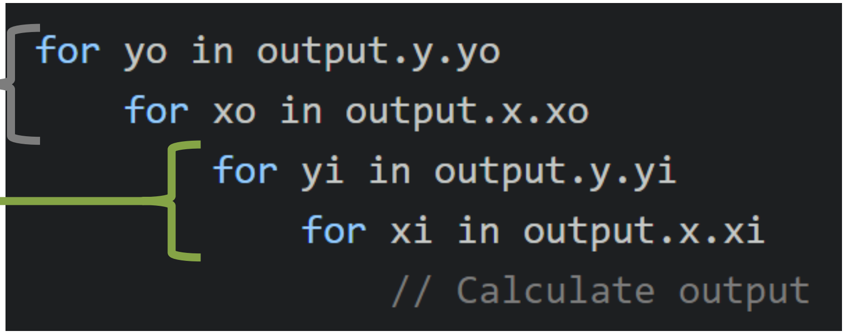


```
for yo in output.y.yo
    for xo in output.x.xo
        for yi in output.y.yi
            for xi in output.x.xi
                // Calculate output
```

# Scheduling a Stage – Tiling (cont.)



Iterates over the tiles

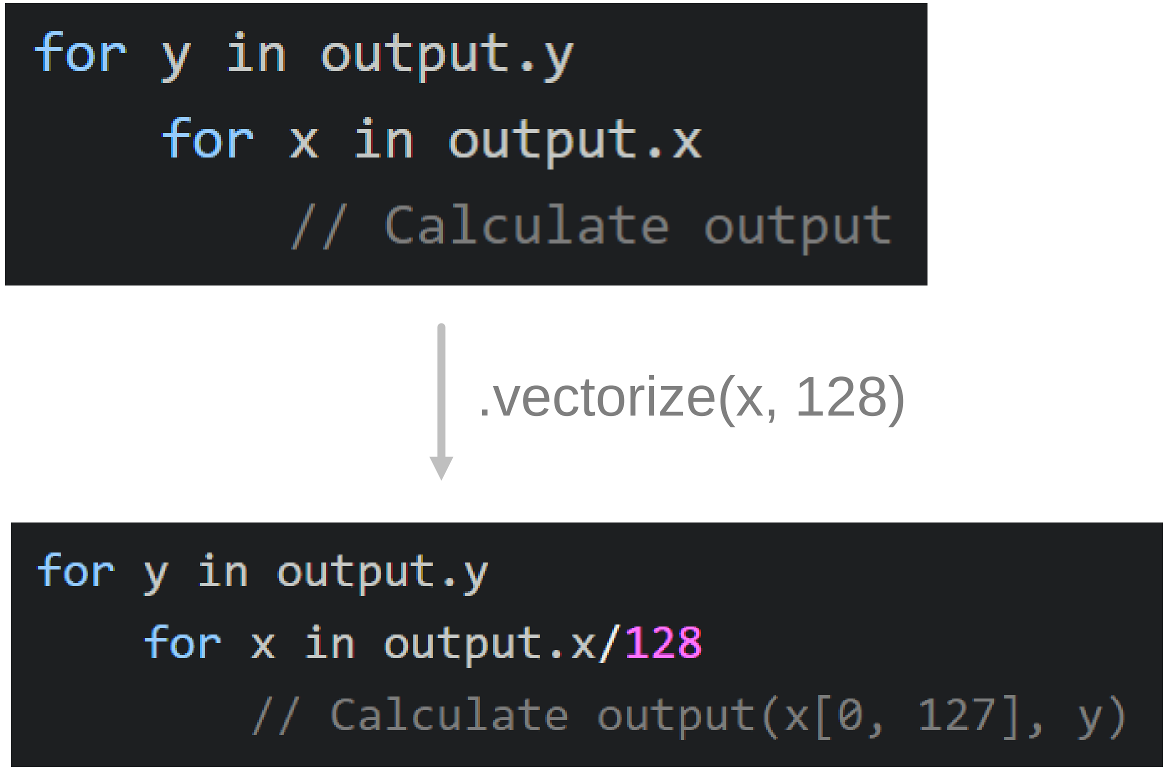


```
for yo in output.y.yo
    for xo in output.x.xo
        for yi in output.y.yi
            for xi in output.x.xi
                // Calculate output
```

Iterates inside the tile

# Scheduling a Stage – Vectorize

- Essentially a split and vectorize
- SIMD instruction set not used until the .vectorize() directive is specified to the stage
- Usage
    - `output.vectorize(x, split_factor)`



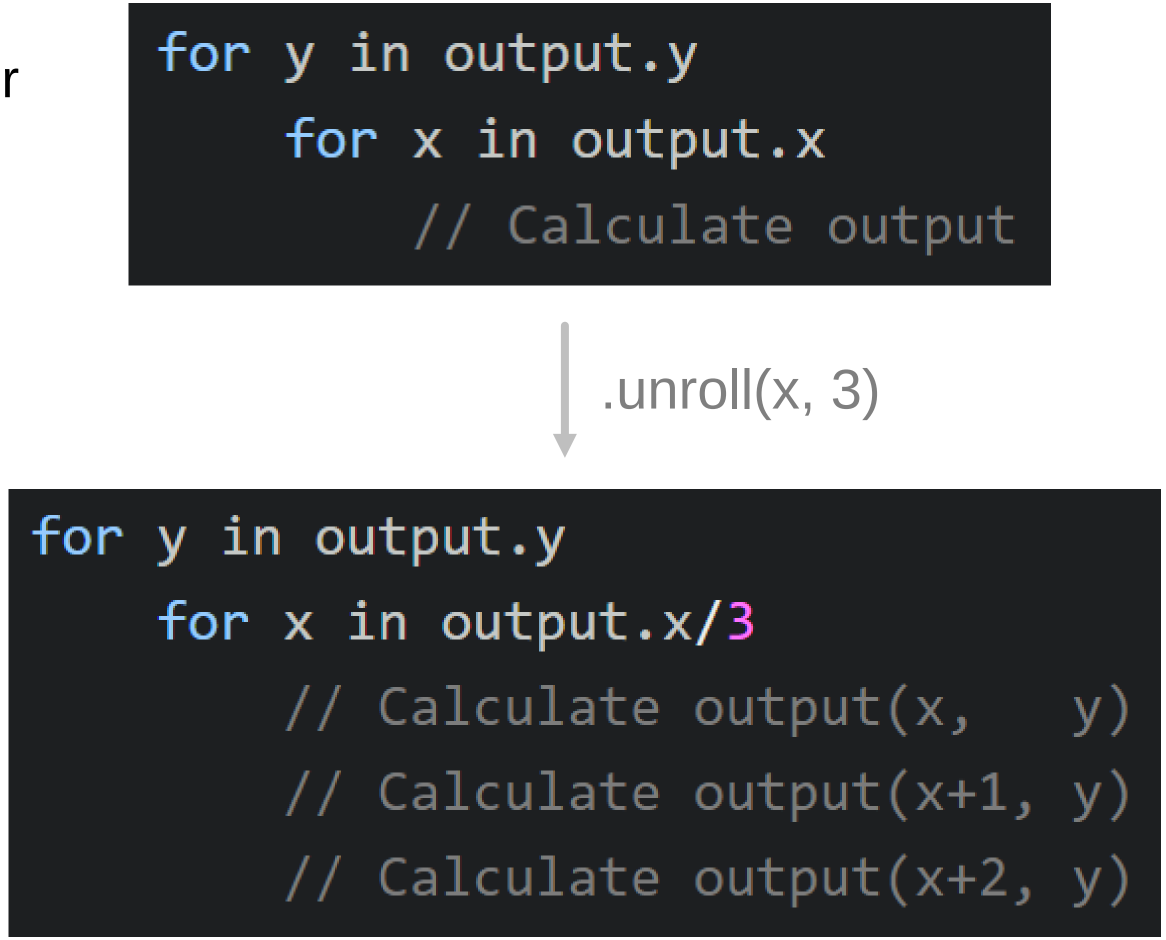
```
for y in output.y
    for x in output.x
        // Calculate output
```

.vectorize(x, 128)

```
for y in output.y
    for x in output.x/128
        // Calculate output(x[0, 127], y)
```

# Scheduling a Stage – Unrolling

- Essentially a split and inner dimension unroll
- Unrolling is important for better register use and better packetization of instructions
- Too much unrolling leads to unnecessary register spills
- `Dimension % unroll_factor = 0` for better performance
- Usage
  - `output.unroll(y, unroll_size)`



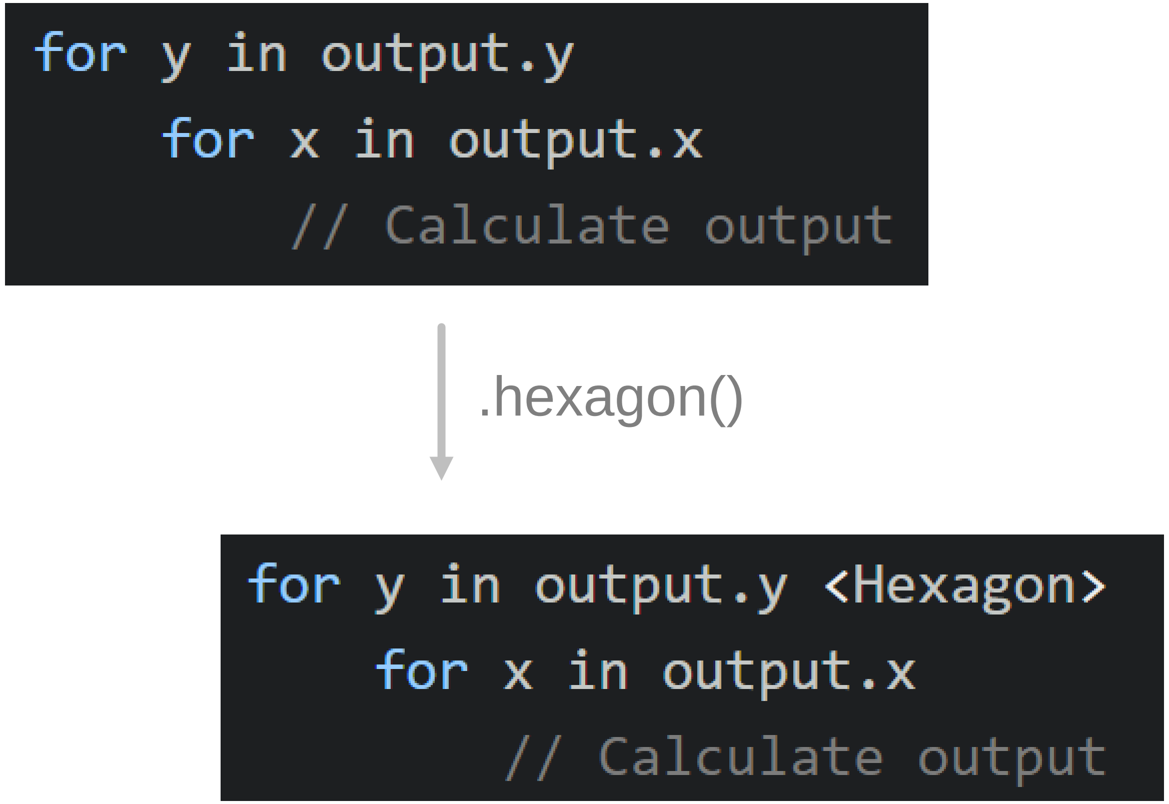
```
for y in output.y
    for x in output.x
        // Calculate output
```

.unroll(x, 3)

```
for y in output.y
    for x in output.x/3
        // Calculate output(x,   y)
        // Calculate output(x+1, y)
        // Calculate output(x+2, y)
```

# Scheduling a Stage – .hexagon()

- Marks the outermost loop
- All stages computed inside the Hexagon marked loop are scheduled on Hexagon
- No manual FastRPC calls to setup data and offload pipeline required
- Usage:
  - `output.hexagon()`



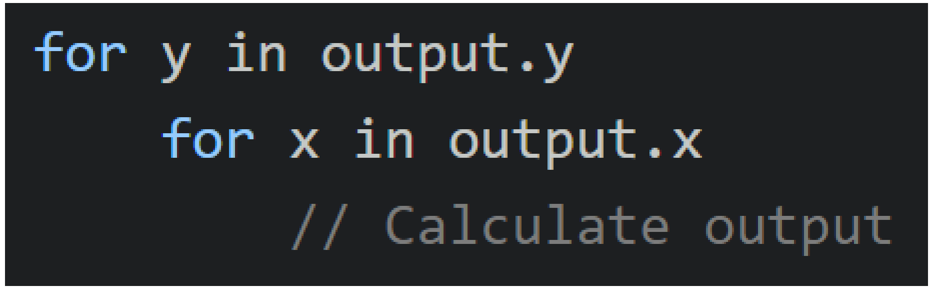
```
for y in output.y
    for x in output.x
        // Calculate output
```

.hexagon()

```
for y in output.y <Hexagon>
    for x in output.x
        // Calculate output
```

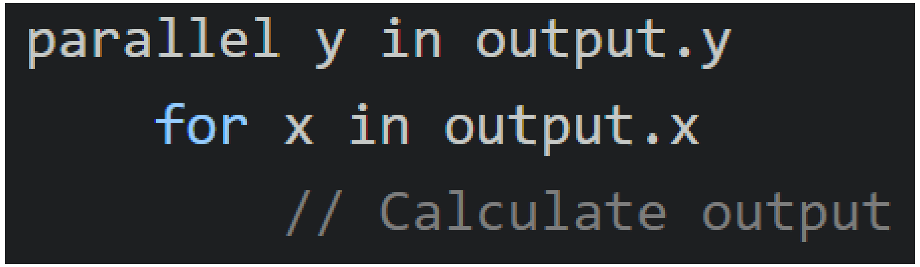
# Scheduling a Stage – Parallel

- Parallelize dimensions across threads
- At the loop position, Halide will dispatch to a thread pool with each thread performing “task_size” amount of work
- Usage:
  - `output.parallel(y, task_size)`



```
for y in output.y
    for x in output.x
        // Calculate output
```

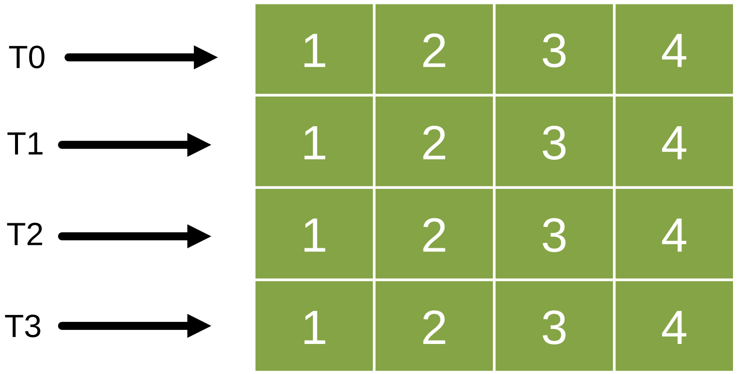
.parallel(y)



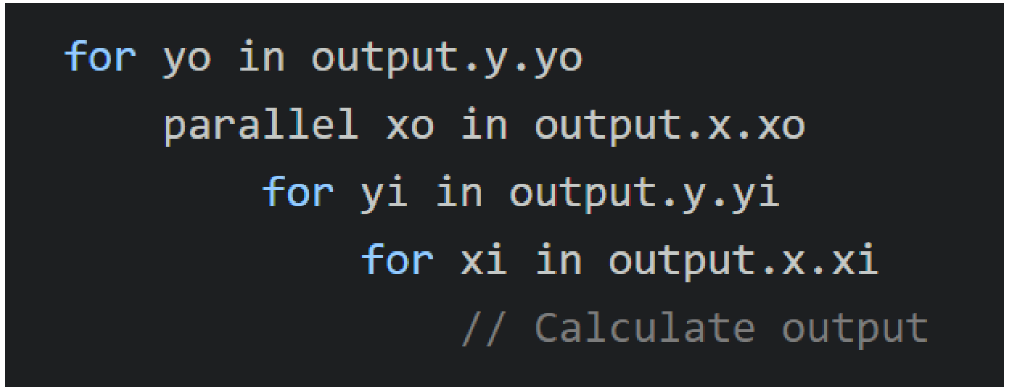
```
parallel y in output.y
    for x in output.x
        // Calculate output
```

4 threads in
thread pool.

task_size = 1 row

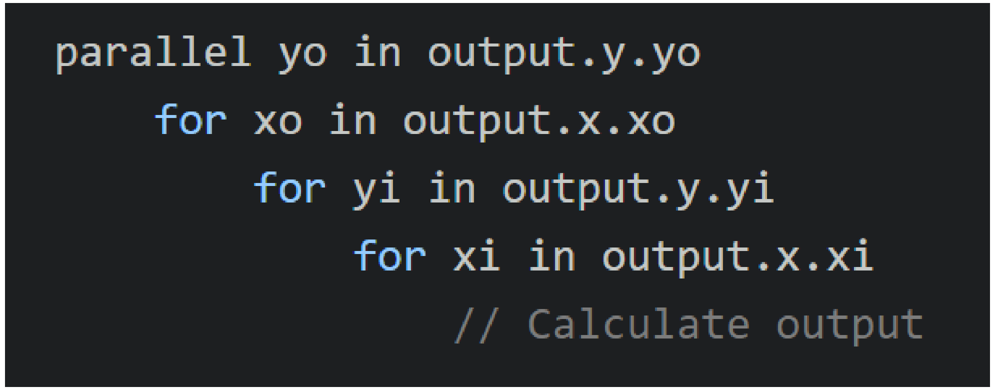


# Scheduling a Stage – Parallelism in Tiling



```
for yo in output.y.yo
    parallel xo in output.x.xo
        for yi in output.y.yi
            for xi in output.x.xi
                // Calculate output
```

```
output
   .tile(x, y, xo, yo, xi, yi, 128, 2)
   .parallel(xo)
```

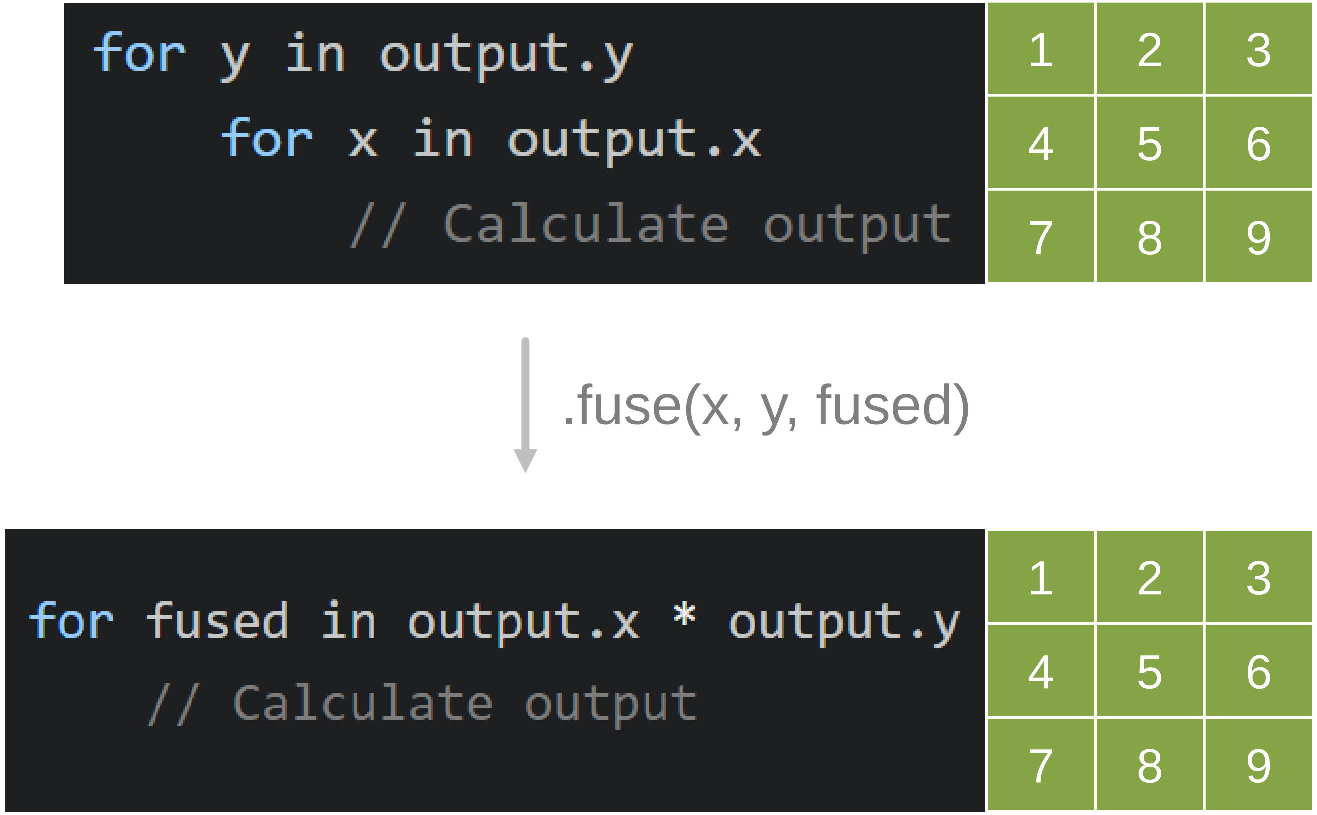


```
parallel yo in output.y.yo
    for xo in output.x.xo
        for yi in output.y.yi
            for xi in output.x.xi
                // Calculate output
```

```
output
   .tile(x, y, xo, yo, xi, yi, 128, 2)
   .parallel(yo)
```

# Scheduling a Stage – Fusing

- Merges two variables (loops) into a single loop over the product of extents
- Usage:
  - `output.fuse(inner_dim, outer_dim, fused_dim)`



```
for y in output.y
    for x in output.x
        // Calculate output
```

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

.fuse(x, y, fused)

```
for fused in output.x * output.y
    // Calculate output
```

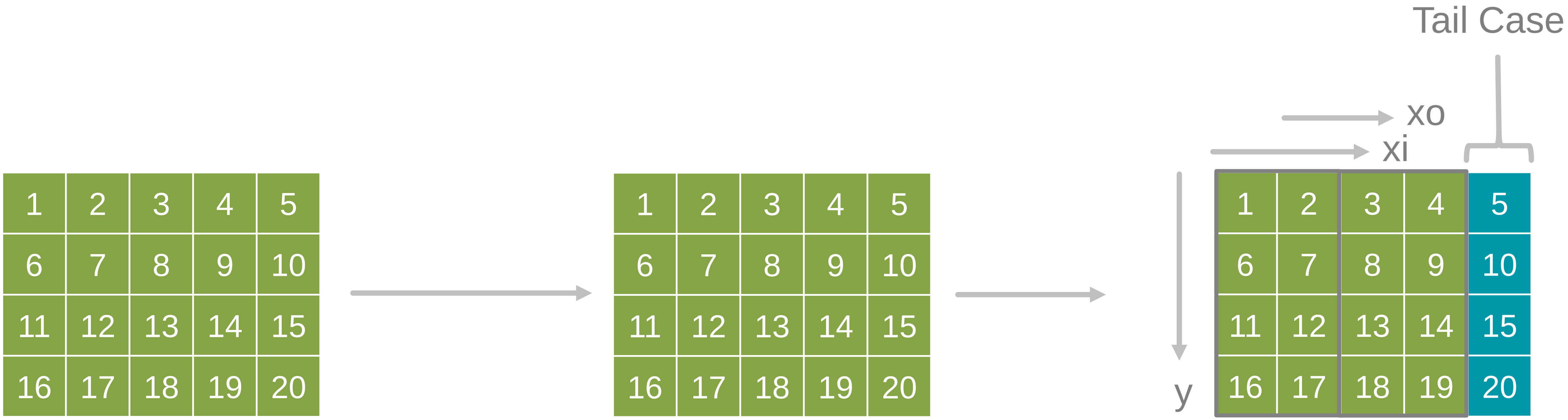
|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

# Scheduling a Stage – Fusing (Why Fuse?)

- Fusing exploits parallelization across dimensions
- Nested parallelism is often slower
- A drawback to fusing is the inability to schedule producer per row of consumer.
  (ie, compute_at(func, y))
- A good example:

```
    output
      .tile(x, y, xi, yi, 128, 2)
      .fuse(x, y, tile_index)
      .parallel(tile_index)
```

# Scheduling a Stage – Revisiting splits



.split(x, xo, xi, 2)

.reorder(xi, y, xo)

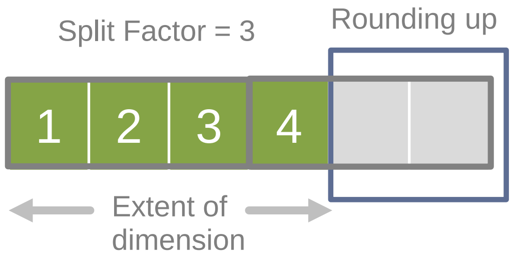
What if the split factor does not divide the size of the dimension?

TailStrategy is important

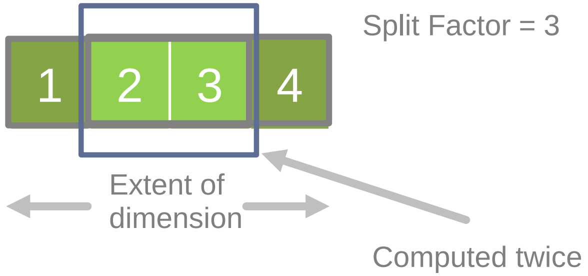
# Scheduling a Stage – TailStrategies

- RoundUp
  - Rounds up the extent to the next multiple of the split factor
  - Simplest and fastest
  - If used on input/output, constrains its size to be a multiple of the split factor
  - For intermediate stages, Halide takes care of ensuring that the dimension size is a multiple of the split factor
- GuardWithIf
  - Guards the inner loop with an if statement to prevent evaluation beyond the original extent
  - Scalarization in the tail case
- ShiftInwards
  - Shifts tail case inward
  - Some redundant evaluation
  - Creates unaligned loads for tail case
  - No scalarization

`TailStrategy::RoundUp`

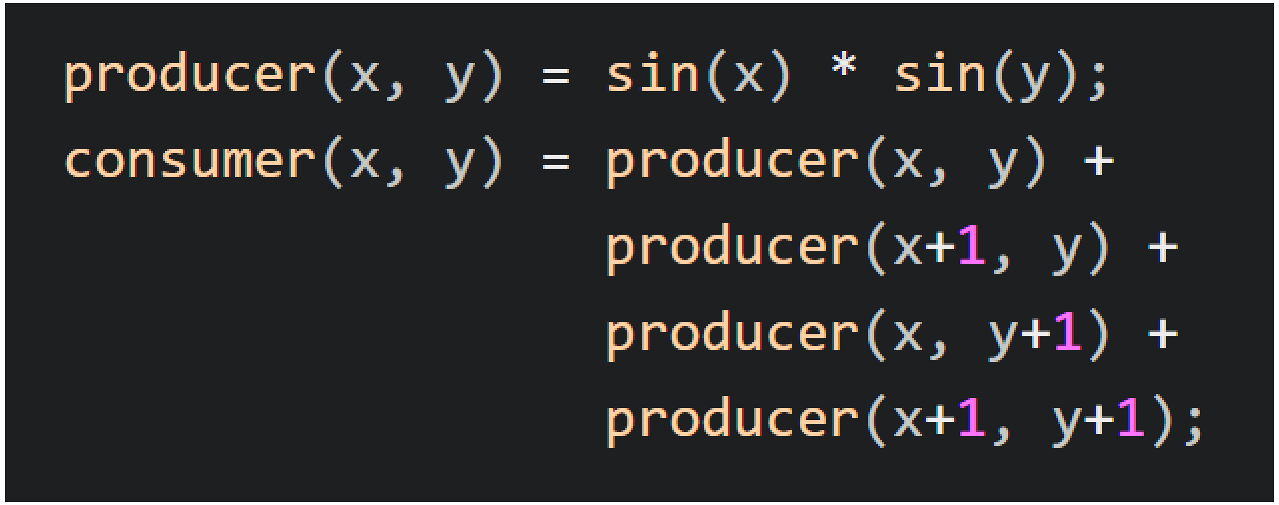


`TailStrategy::ShiftInwards`



# Interleaving Stages – Producer Consumer Relationship

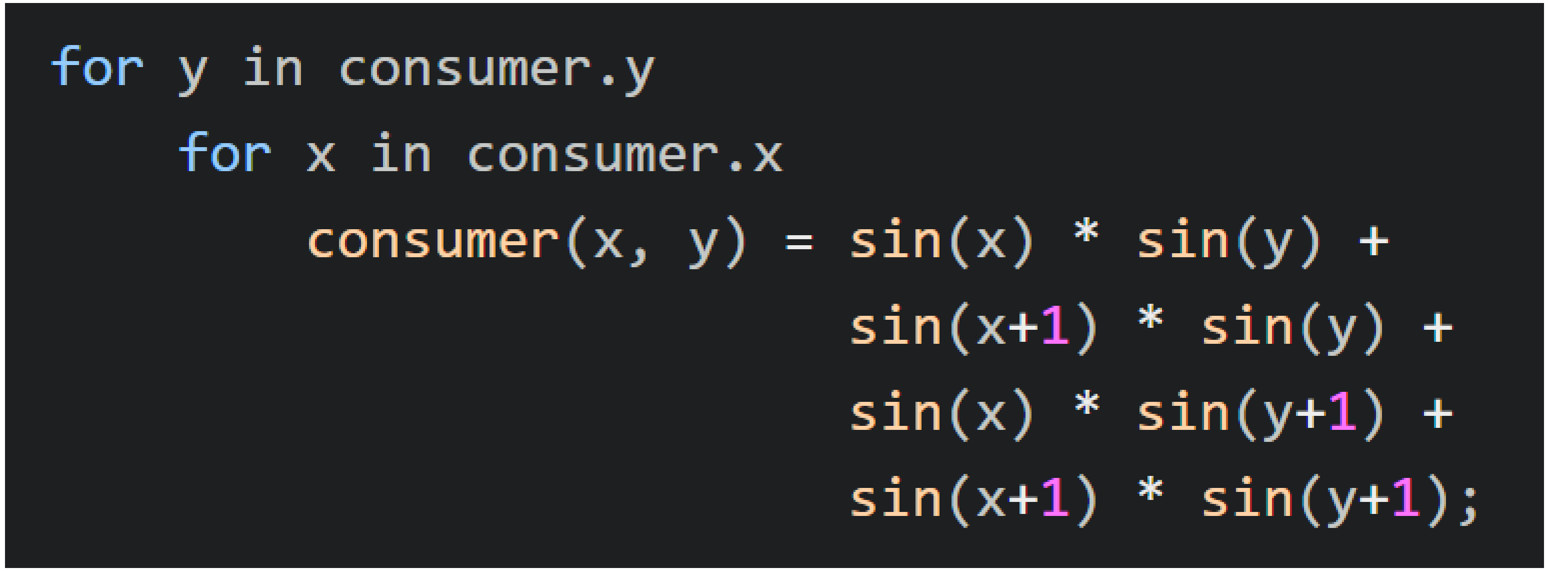
- Consider the Halide pipeline – Consumer feeds on producer values



```
producer(x, y) = sin(x) * sin(y);
consumer(x, y) = producer(x, y) +
                 producer(x+1, y) +
                 producer(x, y+1) +
                 producer(x+1, y+1);
```

# Interleaving Stages – Compute Inline

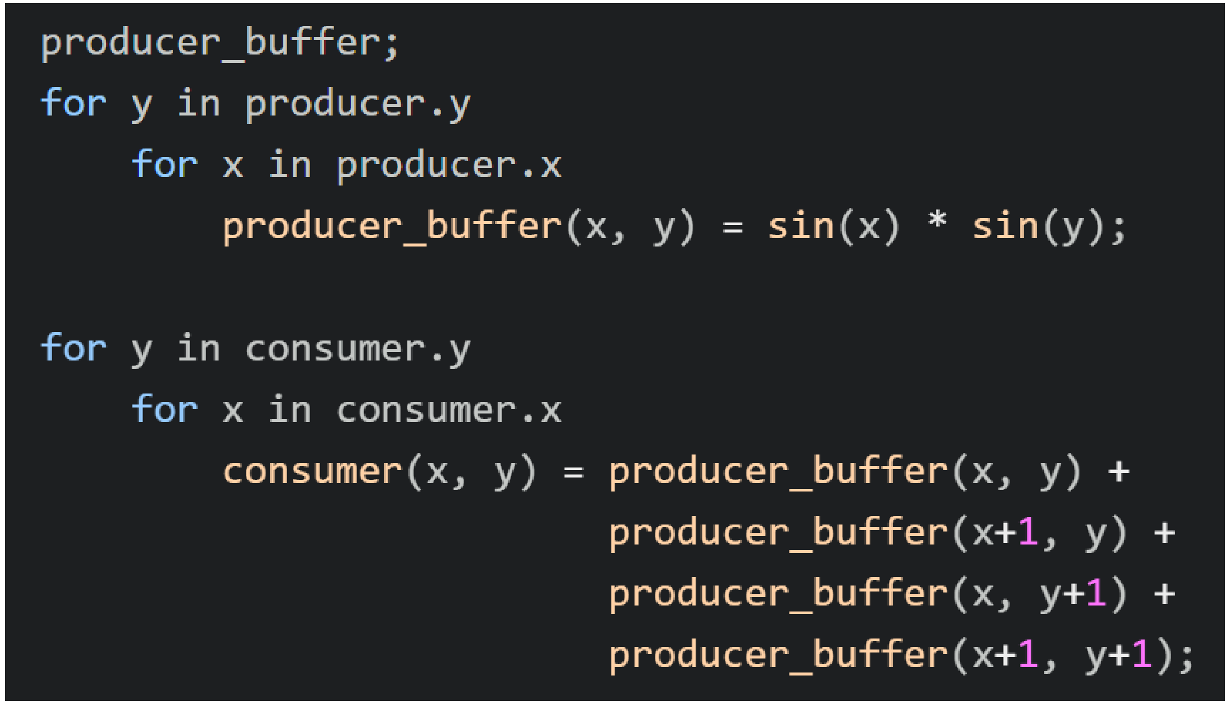
- Calculates the producer as and when required by the consumer
- Directly replaces the producer expression with its value
- By default all stages are inlined



```
for y in consumer.y
    for x in consumer.x
        consumer(x, y) = sin(x) * sin(y) +
                         sin(x+1) * sin(y) +
                         sin(x) * sin(y+1) +
                         sin(x+1) * sin(y+1);
```

# Interleaving Stages – Compute Root

- Calculates entire producer before starting the consumer
- Usage:
  - `producer.compute_root()`



```
producer_buffer;
for y in producer.y
    for x in producer.x
        producer_buffer(x, y) = sin(x) * sin(y);

for y in consumer.y
    for x in consumer.x
        consumer(x, y) = producer_buffer(x, y) +
                         producer_buffer(x+1, y) +
                         producer_buffer(x, y+1) +
                         producer_buffer(x+1, y+1);
```

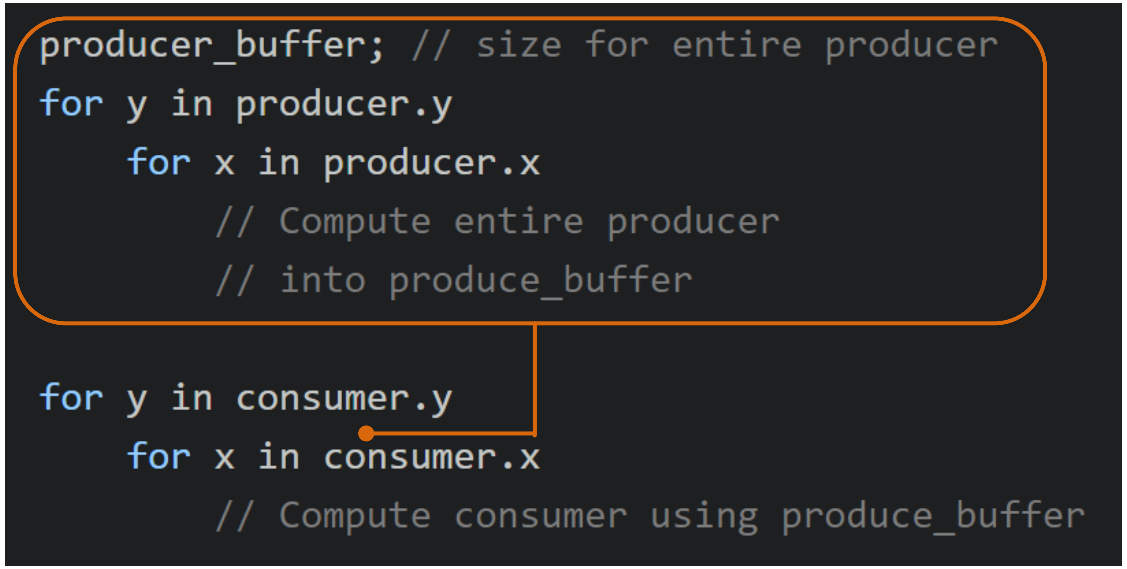
producer_buffer is a Halide created intermediate buffer

# Interleaving Stages – Inline vs. Root

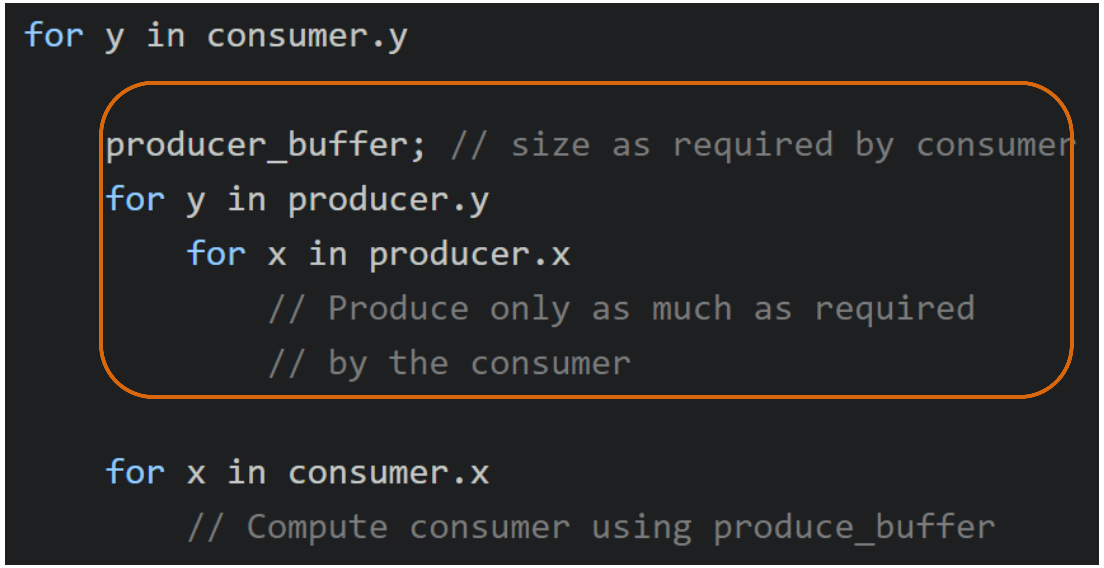
| Compute inline | Compute root |
|---|---|
| Calculates the producer as and when required by the consumer | Calculates all of the producer before calculating any consumer |
| No additional space requirement | Requires extra space |
| Best locality | Poor locality |
| Lots of redundant calculations | No redundant calculations |
| Good for simple stages with less data reuse | Good for complex producer function or if multiple consumers consume the values |

# Interleaving Stages – .compute_at()

- Place the computation of producer inside the specified loop level of the consumer
- Usage
  - `producer.compute_at(consumer, y)`



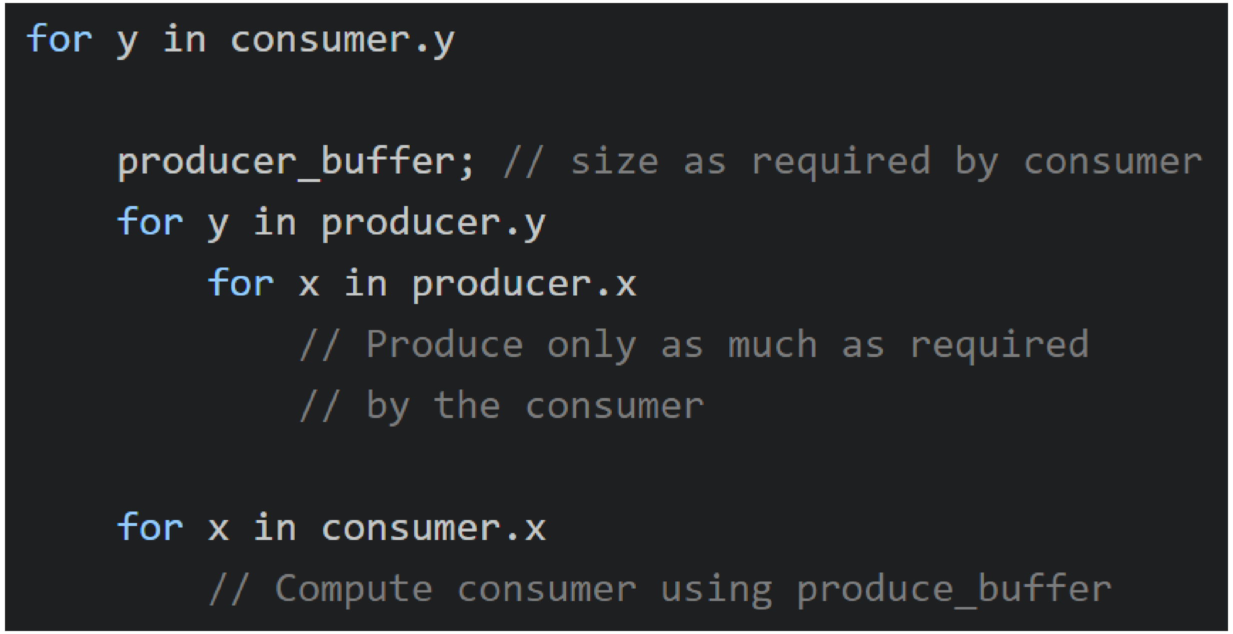
```
producer_buffer; // size for entire producer
for y in producer.y
    for x in producer.x
        // Compute entire producer
        // into produce_buffer

for y in consumer.y
    for x in consumer.x
        // Compute consumer using produce_buffer
```



```
for y in consumer.y

    producer_buffer; // size as required by consumer
    for y in producer.y
        for x in producer.x
            // Produce only as much as required
            // by the consumer

    for x in consumer.x
        // Compute consumer using produce_buffer
```

# Interleaving Stages – .compute_at() (cont.)

- `producer.compute_at(consumer, y);`
- Locality – compute_root < compute_at(consumer, y) < inline
- Redundant computations – compute_root < compute_at(consumer, y) < inline

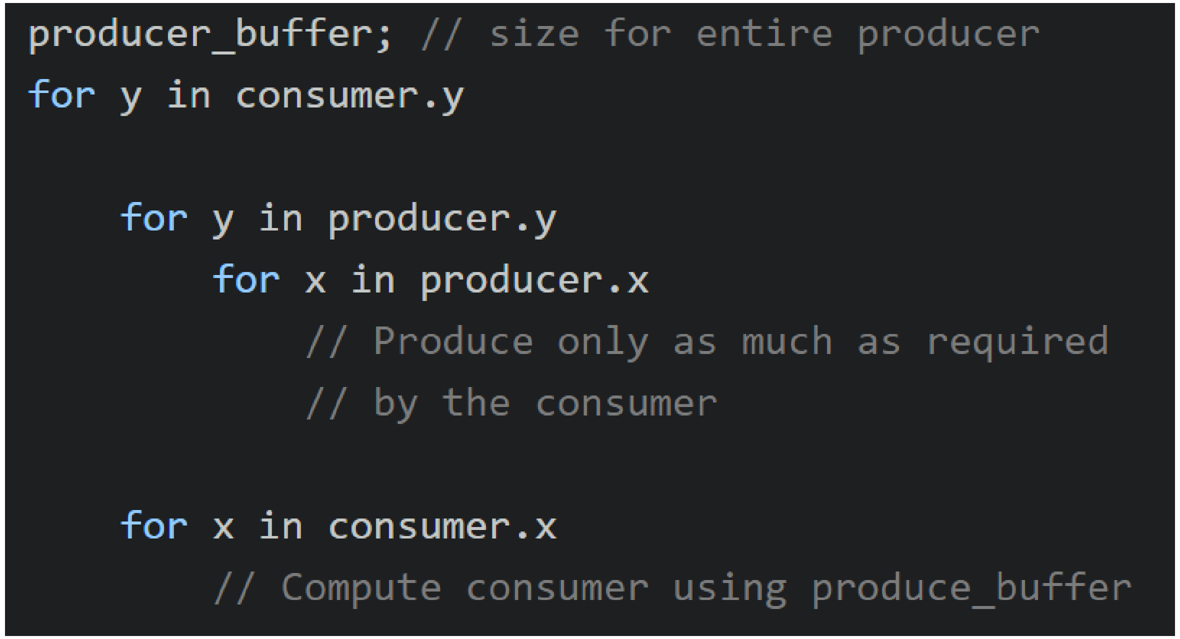


```
for y in consumer.y

    producer_buffer; // size as required by consumer
    for y in producer.y
        for x in producer.x
            // Produce only as much as required
            // by the consumer

    for x in consumer.x
        // Compute consumer using produce_buffer
```

🟧 Bad 🟨 Average 🟩 Good

# Interleaving Stages – .store_at() / .store_root()

- Place the buffer at specified loop level
- By default, storage allocated just before the outermost loop of the stage
- Usage:
  - `producer.store_at(consumer, y), producer.store_root()`



```
producer_buffer; // size for entire producer
for y in consumer.y

    for y in producer.y
        for x in producer.x
            // Produce only as much as required
            // by the consumer

    for x in consumer.x
        // Compute consumer using produce_buffer
```

[redacted]

# Interleaving Stages – Additional Schedules

```
producer.store_root()
          .compute_at(consumer, x);
```

```
consumer.tile(x, y, xi, yi, 4, 4);
producer.compute_at(consumer, x);
```

# Part 3 – Halide for HVX

1

Introduction to Halide

2

The Halide Programming Language

3

Halide for HVX

4

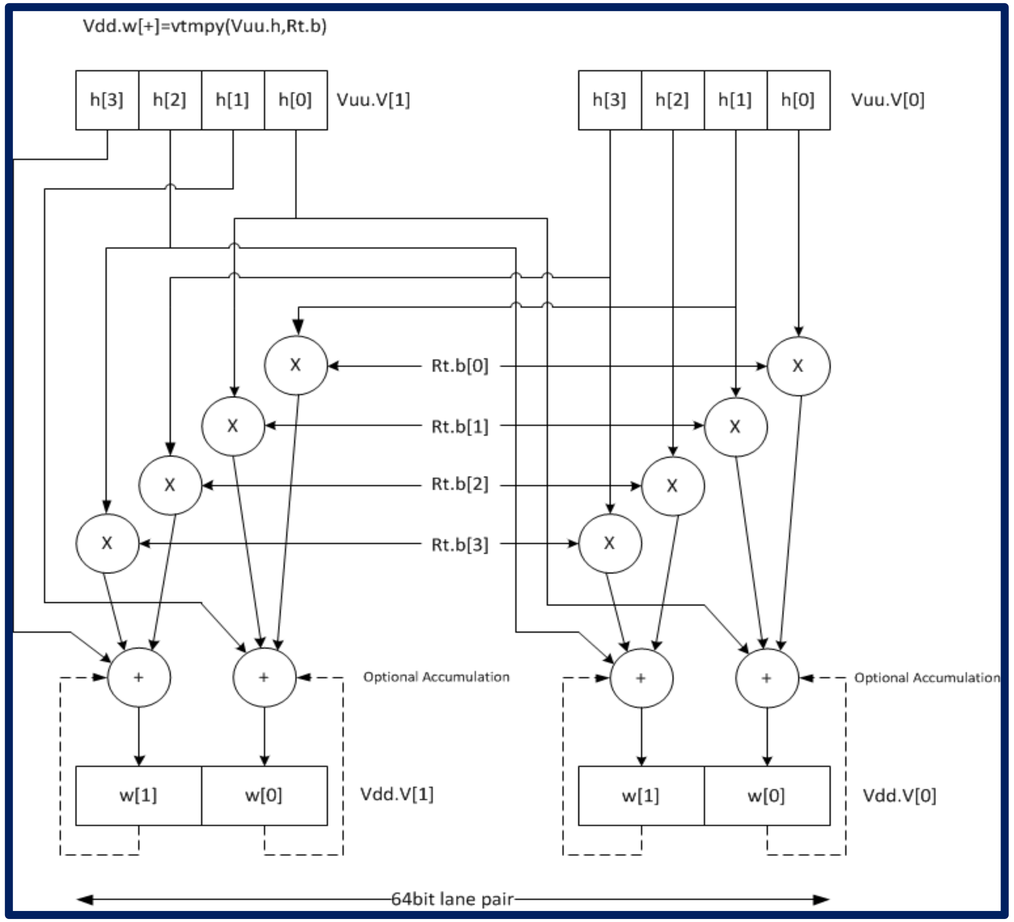
Performance Optimizations

5

Installation Instructions and Troubleshooting

# Halide on Hexagon with HVX – The Need for Halide

- HVX has a large number of instructions (650+)
- HVX is well-suited for image processing
- Complex and distinct DSP flavor
  - Sliding window multiplications
  - Widening multiply accumulate instructions
  - Vector look up table



**Perform a 3-element sliding window pattern operation consisting of a two multiplies with an additional accumulation. Data elements are stored in the vector register pair Vuu, and coefficients in the scalar register Rt**

# Halide on Hexagon with HVX – The Need for Halide (cont.)

- Programming HVX – 3x3 convolution filter in intrinsics

```
static void conv3x3Per2Row(
    unsigned char *restrict inp,
    int          stride,
    int          width,
    signed char   *restrict mask,
    int          shift,
    unsigned char *restrict outp
    )
{
    int i;
    HEXAGON_Vect32 __m2m1m0, __m5m4m3, __m8m7m6;
    HEXAGON_Vect32 *mask4 = (HEXAGON_Vect32 *)mask;

    HVX_Vector sline000,sline004,sline064;
    HVX_Vector sline100,sline104,sline164;
    HVX_Vector sline200,sline204,sline264;
   HVX_Vector sline300,sline304,sline364;
    HVX_Vector sSum20L, sSum31L, sSum20H, sSum31H;
    HVX_Vector sOut01, sOut01p, sOut11, sOut11p, sOut00, sOut10;

    HVX_VectorPair dline000,dline100, dline200,dline300;
    HVX_VectorPair dSum020, dSum031, dSum120, dSum131;

    __m2m1m0 = mask4[0];
    __m5m4m3 = mask4[1];
    __m8m7m6 = mask4[2];

    HVX_Vector *iptr0 = (HVX_Vector *)(inp  - 1*stride);
HVX_Vector *iptr1 = (HVX_Vector *)(inp  + 0*stride);
    HVX_Vector *iptr2 = (HVX_Vector *)(inp  + 1*stride);
    HVX_Vector *iptr3 = (HVX_Vector *)(inp  + 2*stride);
    HVX_Vector *optr0 = (HVX_Vector *)(outp + 0*stride);
    HVX_Vector *optr1 = (HVX_Vector *)(outp + 1*stride);

    sline000 = *iptr0++;
    sline100 = *iptr1++;
    sline200 = *iptr2++;
    sline300 = *iptr3++;

    sOut01p = Q6_V_vzero();
    sOut11p = Q6_V_vzero();

    for ( i=width; i>0; i-=VLEN )
    {
        sline064 = *iptr0++;
        sline164 = *iptr1++;
        sline264 = *iptr2++;
        sline364 = *iptr3++;

        sline004 = Q6_V_valign_VVI(sline064,sline000,4);
        sline104 = Q6_V_valign_VVI(sline164,sline100,4);
        sline204 = Q6_V_valign_VVI(sline264,sline200,4);
        sline304 = Q6_V_valign_VVI(sline364,sline300,4);

        dline000 = Q6_W_vcombine_VV(sline004, sline000);
        dline100 = Q6_W_vcombine_VV(sline104, sline100);
        dline200 = Q6_W_vcombine_VV(sline204, sline200);
        dline300 = Q6_W_vcombine_VV(sline304, sline300);

        dSum020 = Q6_Ww_vrmpy_WubRbI(dline000|
                         ,__m2m1m0, 0);
        dSum031 = Q6_Ww_vrmpy_WubRbI(dline000
                            ,__m2m1m0, 1);

        dSum020 = Q6_Ww_vrmpyacc_WwWubRbI(
         dSum020, dline100,__m5m4m3, 0);
        dSum031 = Q6_Ww_vrmpyacc_WwWubRbI(dSum031, dline100,__m5m4m3, 1);


        dSum020 = Q6_Ww_vrmpyacc_WwWubRbI(dSum020, dline200,__m8m7m6, 0);
        dSum031 = Q6_Ww_vrmpyacc_WwWubRbI(dSum031, dline200,__m8m7m6, 1);

        sSum31L = Q6_Vh_vasr_VwVwR_sat(Q6_V_hi_W(dSum031),Q6_V_lo_W(dSum031),shift);
        sSum20L = Q6_Vh_vasr_VwVwR_sat(Q6_V_hi_W(dSum020),Q6_V_lo_W(dSum020),shift);
        sOut01 = Q6_Vub_vsat_VhVh(sSum31L,sSum20L);
        sOut00 = Q6_V_vlalign_VVI(sOut01,sOut01p,1);
        *optr0++ = sOut00;

        dSum120 = Q6_Ww_vrmpy_WubRbI(dline100,__m2m1m0, 0);
        dSum131 = Q6_Ww_vrmpy_WubRbI(dline100,__m2m1m0, 1);

        dSum120 = Q6_Ww_vrmpyacc_WwWubRbI(dSum120, dline200,__m5m4m3, 0);
        dSum131 = Q6_Ww_vrmpyacc_WwWubRbI(dSum131, dline200,__m5m4m3, 1);

        dSum120 = Q6_Ww_vrmpyacc_WwWubRbI(dSum120, dline300,__m8m7m6, 0);
        dSum131 = Q6_Ww_vrmpyacc_WwWubRbI(dSum131, dline300,__m8m7m6, 1);

        sSum31H = Q6_Vh_vasr_VwVwR_sat(Q6_V_hi_W(dSum131),Q6_V_lo_W(dSum131),shift);
…
```

# Halide on Hexagon with HVX – The Need for Halide (cont.)

- Programming HVX – 3x3 convolution filter in intrinsics

```
static void conv3x3Per2Row
    unsigned char *restrict inp
    int           stride,
    int           width,
    signed char   *restrict mas
    int           shift,
    unsigned char *restrict out
    )
{
    int i;
    HEXAGON_Vect32 __m2
8m7m6;
    HEXAGON_Vect32 *mask
*)mask;

    HVX_Vector sline000,sline
    HVX_Vector sline100,sline
    HVX_Vector sline200,sline
 HVX_Vector sline300,sline3
```

```
_Vh_vasr_VwVwR_sat(Q6_V_hi_W(d
dSum031),shift);
6_Vh_vasr_VwVwR_sat(Q6_V_hi_W(
(dSum020),shift);
Vub_vsat_VhVh(sSum31L,sSum20L)

V_vlalign_VVI(sOut01,sOut01p,1);
ut00;

6_Ww_vrmpy_WubRbI(dline100,__m

6_Ww_vrmpy_WubRbI(dline100,__m

6_Ww_vrmpyacc_WwWubRbI(dSum
m3, 0);
6_Ww_vrmpyacc_WwWubRbI(dSum
m3, 1);

6_Ww_vrmpyacc_WwWubRbI(dSum
m6, 0);
6_Ww_vrmpyacc_WwWubRbI(dSum
m6, 1);

6_Vh_vasr_VwVwR_sat(Q6_V_hi_W
/(dSum131),shift);
```

Simple filter – 3x3 convolution

130 lines of highly optimized HVX intrinsics

Very high level of expertise required

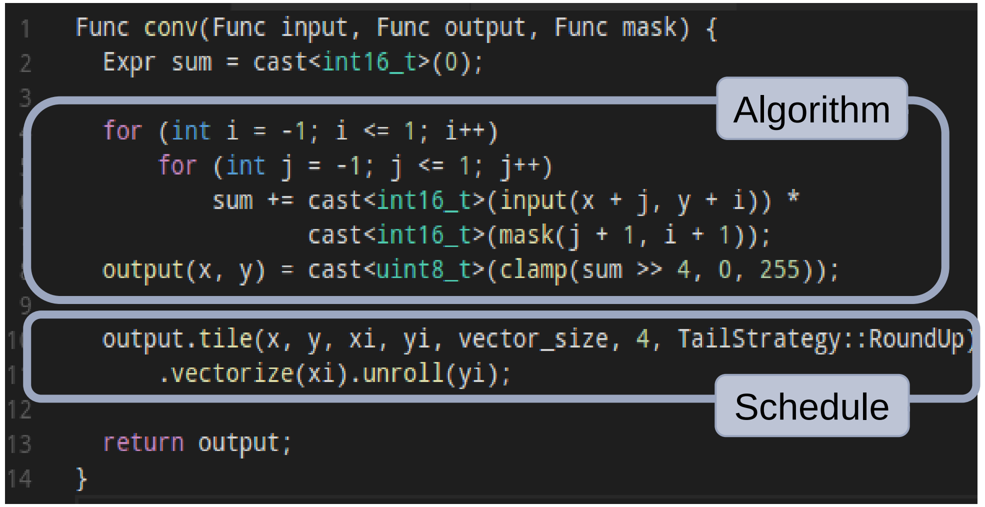
Requires deep knowledge of HVX architecture and microarchitecture

# Halide on Hexagon with HVX – Halide vs. HVX Intrinsics

- 3x3 convolution

## Halide

0.109 cycles/pixel

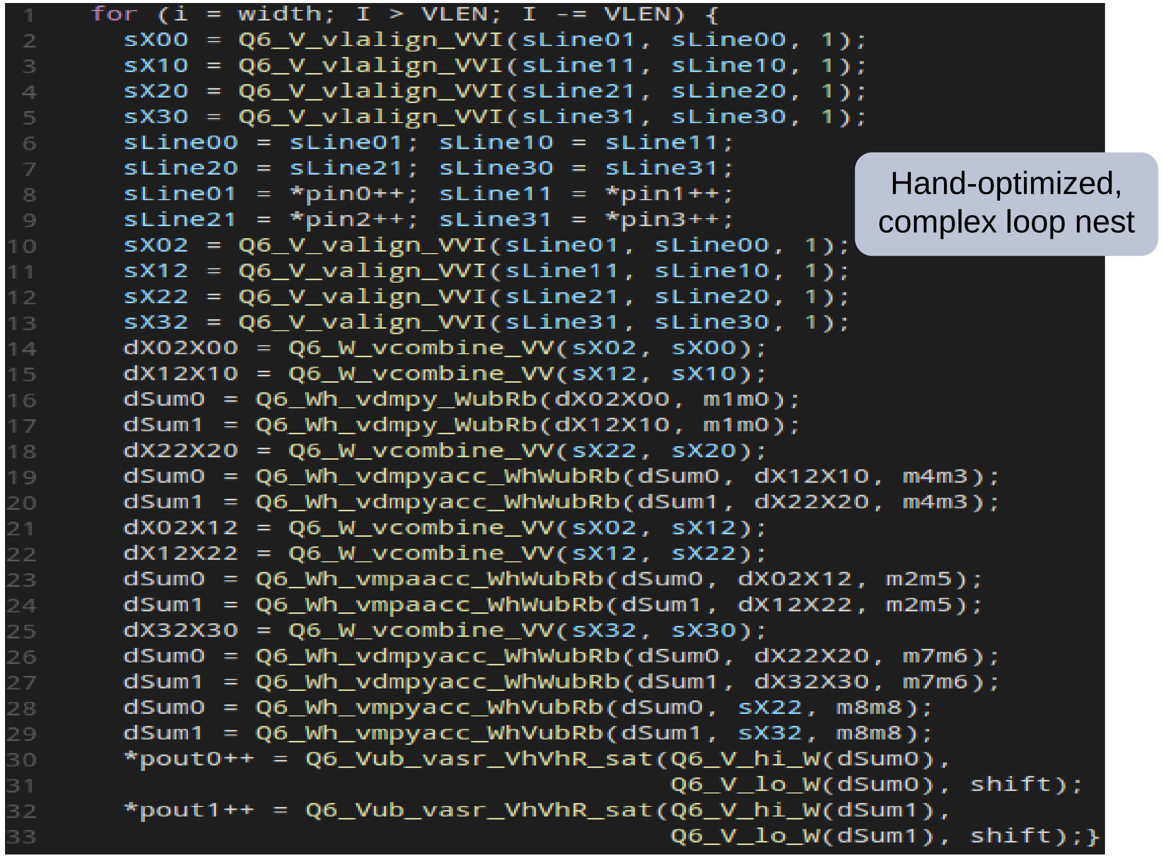


```
Func conv(Func input, Func output, Func mask) {
  Expr sum = cast<int16_t>(0);

  for (int i = -1; i <= 1; i++)
      for (int j = -1; j <= 1; j++)
          sum += cast<int16_t>(input(x + j, y + i)) *
                 cast<int16_t>(mask(j + 1, i + 1));
  output(x, y) = cast<uint8_t>(clamp(sum >> 4, 0, 255));

  output.tile(x, y, xi, yi, vector_size, 4, TailStrategy::RoundUp)
      .vectorize(xi).unroll(yi);

  return output;
}
```

- **Halide version – 14 lines**
- **Very concise, easy to maintain**
- **Developers can quickly restructure schedule to optimize the algorithm**
  - **Corresponding optimization in intrinsics version can be extremely complex**

## HVX Intrinsics

0.114 cycles/pixel



```
for (i = width; I > VLEN; I -= VLEN) {
    sX00 = Q6_V_vlalign_VVI(sLine01, sLine00, 1);
    sX10 = Q6_V_vlalign_VVI(sLine11, sLine10, 1);
    sX20 = Q6_V_vlalign_VVI(sLine21, sLine20, 1);
    sX30 = Q6_V_vlalign_VVI(sLine31, sLine30, 1);
    sLine00 = sLine01; sLine10 = sLine11;
    sLine20 = sLine21; sLine30 = sLine31;
    sLine01 = *pin0++; sLine11 = *pin1++;
    sLine21 = *pin2++; sLine31 = *pin3++;
    sX02 = Q6_V_valign_VVI(sLine01, sLine00, 1);
    sX12 = Q6_V_valign_VVI(sLine11, sLine10, 1);
    sX22 = Q6_V_valign_VVI(sLine21, sLine20, 1);
    sX32 = Q6_V_valign_VVI(sLine31, sLine30, 1);
    dX02X00 = Q6_W_vcombine_VV(sX02, sX00);
    dX12X10 = Q6_W_vcombine_VV(sX12, sX10);
    dSum0 = Q6_Wh_vdmpy_WubRb(dX02X00, m1m0);
    dSum1 = Q6_Wh_vdmpy_WubRb(dX12X10, m1m0);
    dX22X20 = Q6_W_vcombine_VV(sX22, sX20);
    dSum0 = Q6_Wh_vdmpyacc_WhWubRb(dSum0, dX12X10, m4m3);
    dSum1 = Q6_Wh_vdmpyacc_WhWubRb(dSum1, dX22X20, m4m3);
    dX02X12 = Q6_W_vcombine_VV(sX02, sX12);
    dX12X22 = Q6_W_vcombine_VV(sX12, sX22);
    dSum0 = Q6_Wh_vmpaacc_WhWubRb(dSum0, dX02X12, m2m5);
    dSum1 = Q6_Wh_vmpaacc_WhWubRb(dSum1, dX12X22, m2m5);
    dX32X30 = Q6_W_vcombine_VV(sX32, sX30);
    dSum0 = Q6_Wh_vdmpyacc_WhWubRb(dSum0, dX22X20, m7m6);
    dSum1 = Q6_Wh_vdmpyacc_WhWubRb(dSum1, dX32X30, m7m6);
    dSum0 = Q6_Wh_vmpyacc_WhVubRb(dSum0, sX22, m8m8);
    dSum1 = Q6_Wh_vmpyacc_WhVubRb(dSum1, sX32, m8m8);
    *pout0++ = Q6_Vub_vasr_VhVhR_sat(Q6_V_hi_W(dSum0),
                                     Q6_V_lo_W(dSum0), shift);
    *pout1++ = Q6_Vub_vasr_VhVhR_sat(Q6_V_hi_W(dSum1),
                                     Q6_V_lo_W(dSum1), shift);}
```

Hand-optimized, complex loop nest

***…94 more lines of  HVX intrinsics code not shown***

# Halide on Hexagon with HVX

- Key Takeaway from Examples
  - To program a simple imaging filter in intrinsics or assembly requires a large degree of expertise in:
    - HVX architecture – Knowledge of HVX instruction set and optimal packetization of instructions
    - HVX microarchitecture – Latency of instructions, why stalls are caused, latency of prefetch, cache, etc.
    - Expertise in image algorithms
- Halide's goal is to significantly reduce the expertise needed in HVX architecture and microarchitecture
- Halide allows programmers to focus development effort on image algorithms, thereby improving programmer productivity

Halide improvement means:

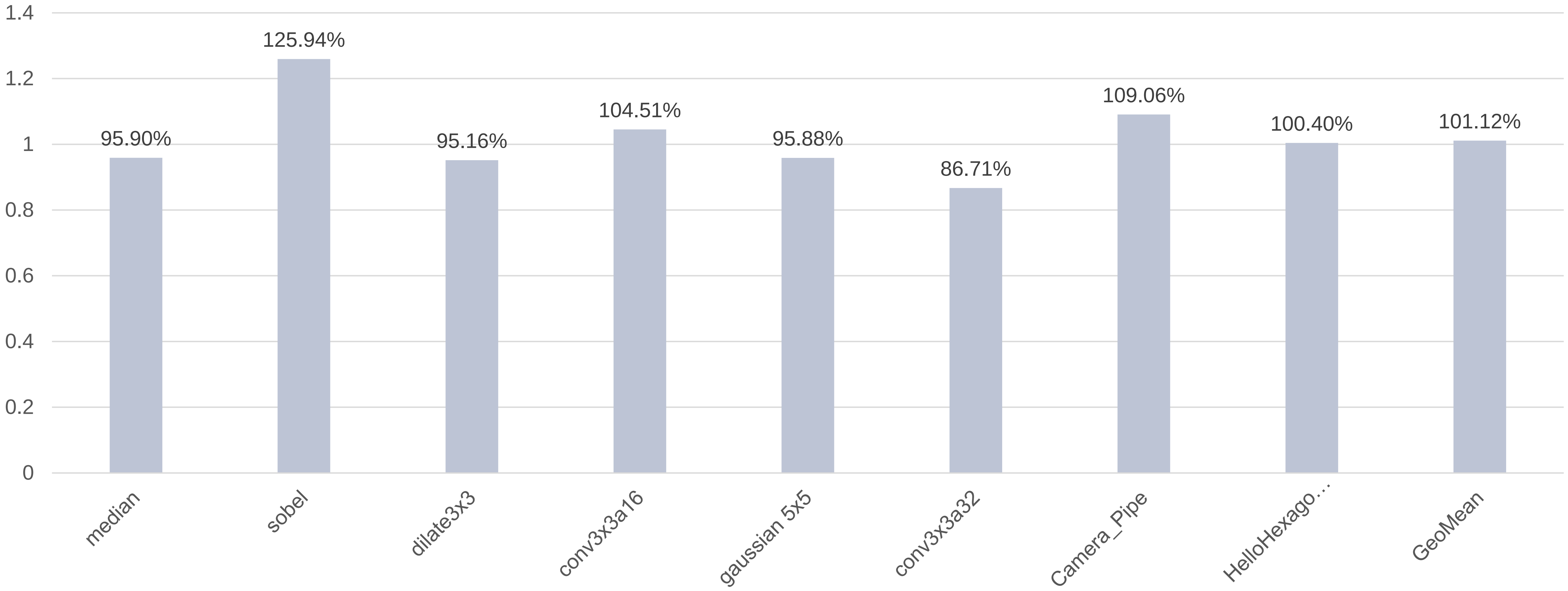
- Ability to write the algorithm once, test and freeze the algorithm after the tests pass.
- Then, alter schedules to explore optimizations to improve performance.
- Potential to never change the algorithm again.

# Advantages of Halide vs. Intrinsics or Assembly

- Development time and cost
  - Significantly more time needed to author assembly/intrinsics compared to a Halide
- Expertise
  - Assembly/intrinsics require nuanced understanding of processor architecture and micro-architecture
    - Only scales to a few programmers
  - Halide allows a significantly larger set of developers to use HVX
  - Halide allows programmers to focus on the algorithm and not on processor details
- Scalability
  - Assembly and intrinsics programming does not scale to very large code bases
- Automatic optimization
  - Halide allows optimizations across kernels for the entire image pipeline
    - Not possible to optimize across libraries of assembly/intrinsics
  - Halide allows programmers to easily tune program for performance
- Portability
  - With Halide, no change in code required for new versions of Hexagon
  - Compiler automatically takes advantage of new hardware features and instruction sets

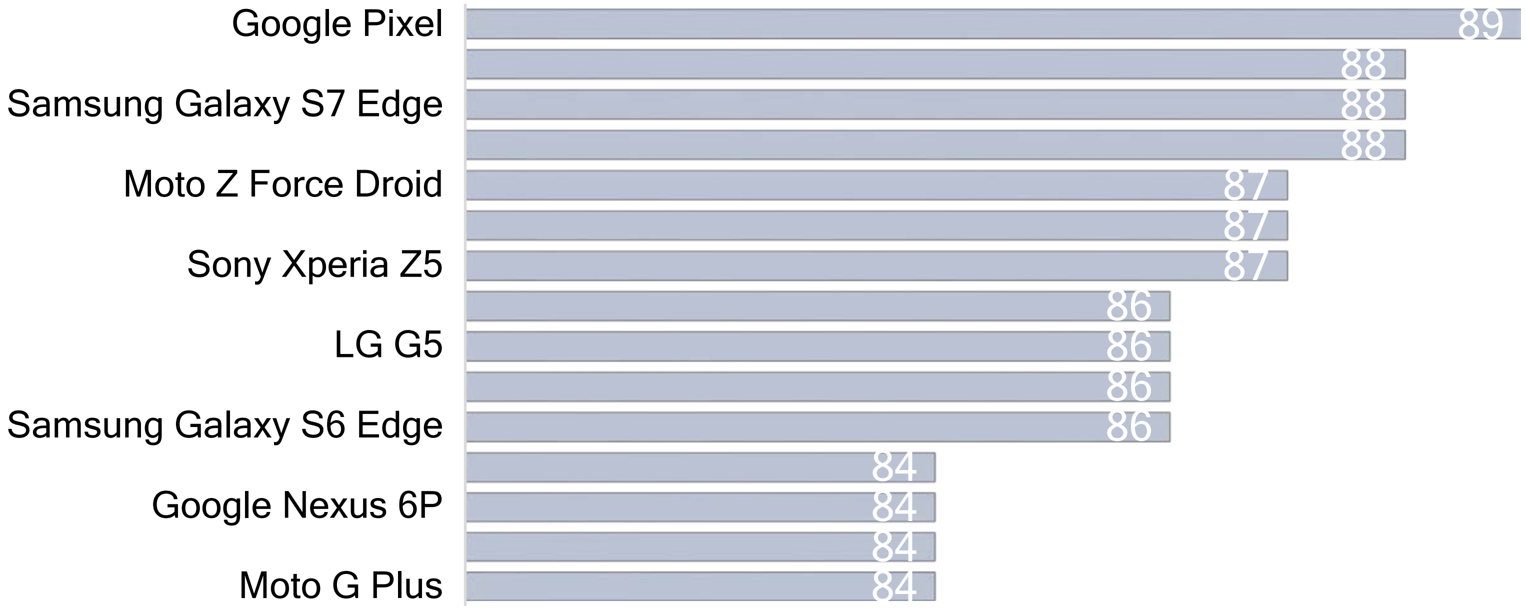
# Halide HVX v60 Performance on Hexagon Simulator

- Halide performance vs. hand-optimized intrinsic benchmarks
- Higher is better
- 100% equals intrinsic performance



# Products Launched with Halide on HVX

- Google Pixel – Launched in Oct 2016
  - Camera was the feature highlighted by Google at launch
  - Achieved Highest DxOMark rating at Pixel launch time (by any phone ever)
  - Achieved using Halide on HVX to implement a significant part of the camera pipeline



# Halide on Hexagon with HVX

- HVX has two execution modes, 64 byte and 128 byte vector length modes
  - 128B mode is will be the only supported mode in future chips
- Halide compiler for HVX supports both modes which can be selected by way of a target feature
- Supported on all QTI chipsets that contain HVX

# Halide on Hexagon with HVX – Execution Modes

- 4 available execution modes to accommodate simulation and on-target environments
- Offload modes – Device-Offload and Simulator-Offload
  - Part of the entire pipeline can be executed on Hexagon (hardware or simulator) and the rest on ARM (device-offload) or on x86 (simulator-offload)
  - To offload, append the “.hexagon()” scheduling directive to a pipeline,
    - dilate3x3.hexagon().vectorize(xi, 128);
- Standalone modes – Device-Standalone and Simulator-Standalone
  - The entire pipeline is executed on Hexagon hardware (Device-standalone) or Hexagon simulator (Simulator-standalone)
  - Differ from the offload models in that the runtimes that support the Halide pipelines are minimal and do not use FastRPC
  - Simulator-standalone allows developers to prototype future hardware versions and features
  - Enable integration with pipeline stages not written in Halide
  - Do not need the “.hexagon()” scheduling directive
    - dilate3x3~~.hexagon()~~.vectorize(xi, 128);

# Execution Modes – Determining Which Mode to Use

- I want to write a new image processing application for a handheld device. I am considering writing it in Halide. I want to run it on ARM, but some part of my algorithm can be written in fixed point and is performance critical and so I could use HVX for it. But, I have never used FastRPC before. Which mode should I use?

Start with Simulator-Offload mode. Install the Hexagon SDK. Write your application and after you have decided which stages to offload to Hexagon, test it in Simulator-Offload mode. Once satisfied, deploy your application on the device and use the Device-Offload mode

- I have an application that runs on a Snapdragon™ MTP/Dragonboard. Some parts of it run on the Hexagon DSP with HVX already. However, code written in Halide is more maintainable and easier to optimize for performance, so I would like to rewrite these parts in Halide. I am an expert in FastRPC and my application uses FastRPC already. I am also willing to take care of clock voting and setting HVX power mode etc. Which mode should I use?

Start with Simulator-Standalone mode. Install the Hexagon SDK. Transition the intended part of your application to Halide and test with synthetic input (if possible) in Simulator-Standalone mode. Once satisfied, compile your pipeline in Device-Standalone mode and link into your application and run on device

# Execution Modes – Determining Which Mode to Use (cont.)

- I want to write a new image processing application for a handheld device. I am considering writing it in Halide. I want to run it on ARM, but some part of my algorithm can be written in fixed point and is performance critical and so I could use HVX for it. But, I have never used FastRPC before. Which mode should I use?

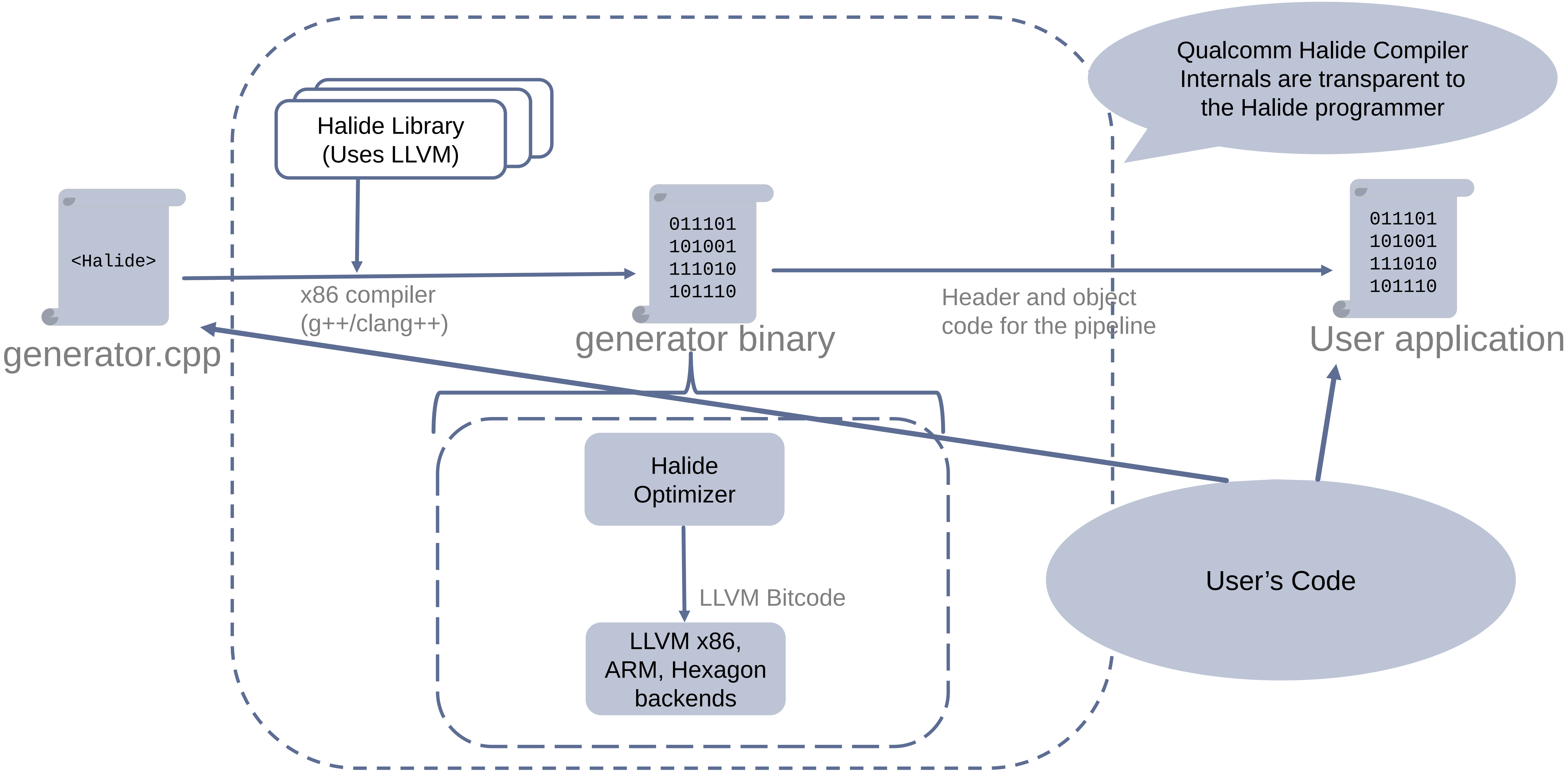
Start with Simulator-Off … after you have decided which
stages to offl … ication on the device
and use …

**Remember!**
In the two Offload modes, the Halide compiler takes care of code and data transfer between the host CPU and the Hexagon DSP using FastRPC (Device-Offload) or message passing (Simulator-Offload). In the two Standalone modes, it is the programmer’s responsibility to take care of the communication between the host CPU and the Hexagon DSP.

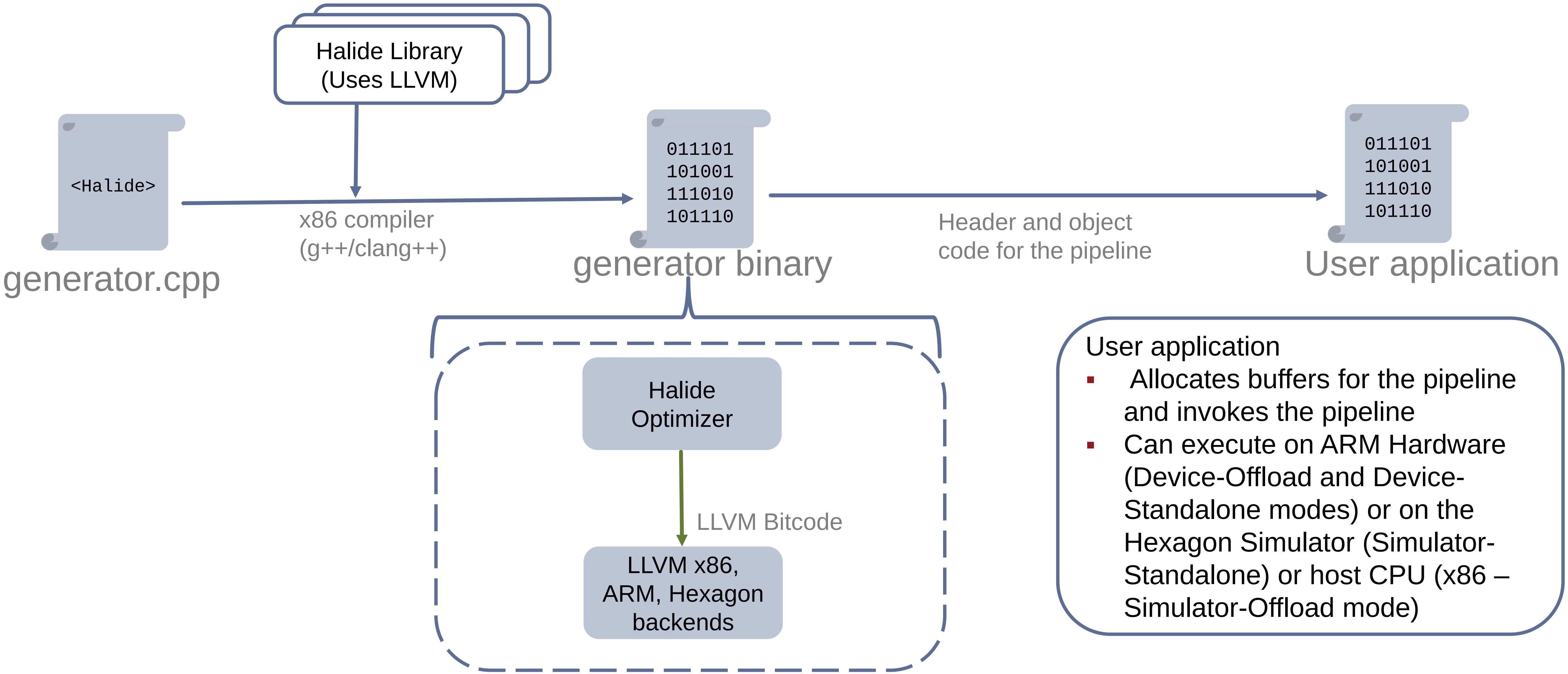
- I h … SP with
HV … o I
wou … eady. I am
also wi …

Start with Simulator-Stand … ed part of your application to
Halide and test with synthetic input (if possible) in Simulator-Standalone mode. Once satisfied, compile your pipeline in Device-Standalone mode and link into your application and run on device

# Halide Compiler – Ahead-of-Time (AOT) Compilation


Qualcomm Halide Compiler Internals are transparent to the Halide programmer
Halide Library (Uses LLVM)
<Halide>
generator.cpp
x86 compiler (g++/clang++)
011101 101001 111010 101110
generator binary
Header and object code for the pipeline
011101 101001 111010 101110
User application
Halide Optimizer
LLVM Bitcode
LLVM x86, ARM, Hexagon backends
User's Code

# Halide Compiler – Ahead-of-Time (AOT) Compilation (cont.)



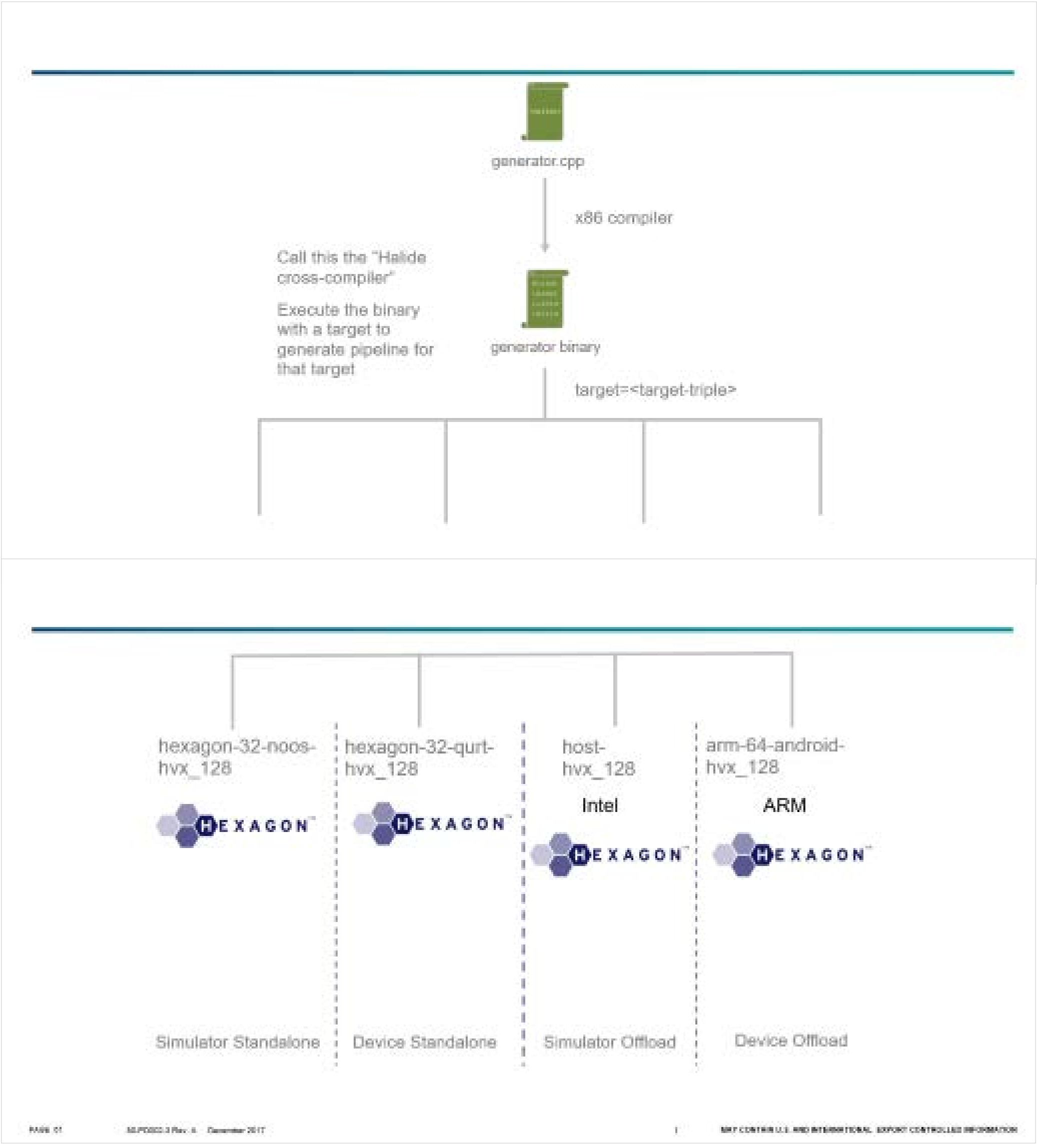
User application

- Allocates buffers for the pipeline and invokes the pipeline
- Can execute on ARM Hardware (Device-Offload and Device-Standalone modes) or on the Hexagon Simulator (Simulator-Standalone) or host CPU (x86 – Simulator-Offload mode)

# Compilation Stages

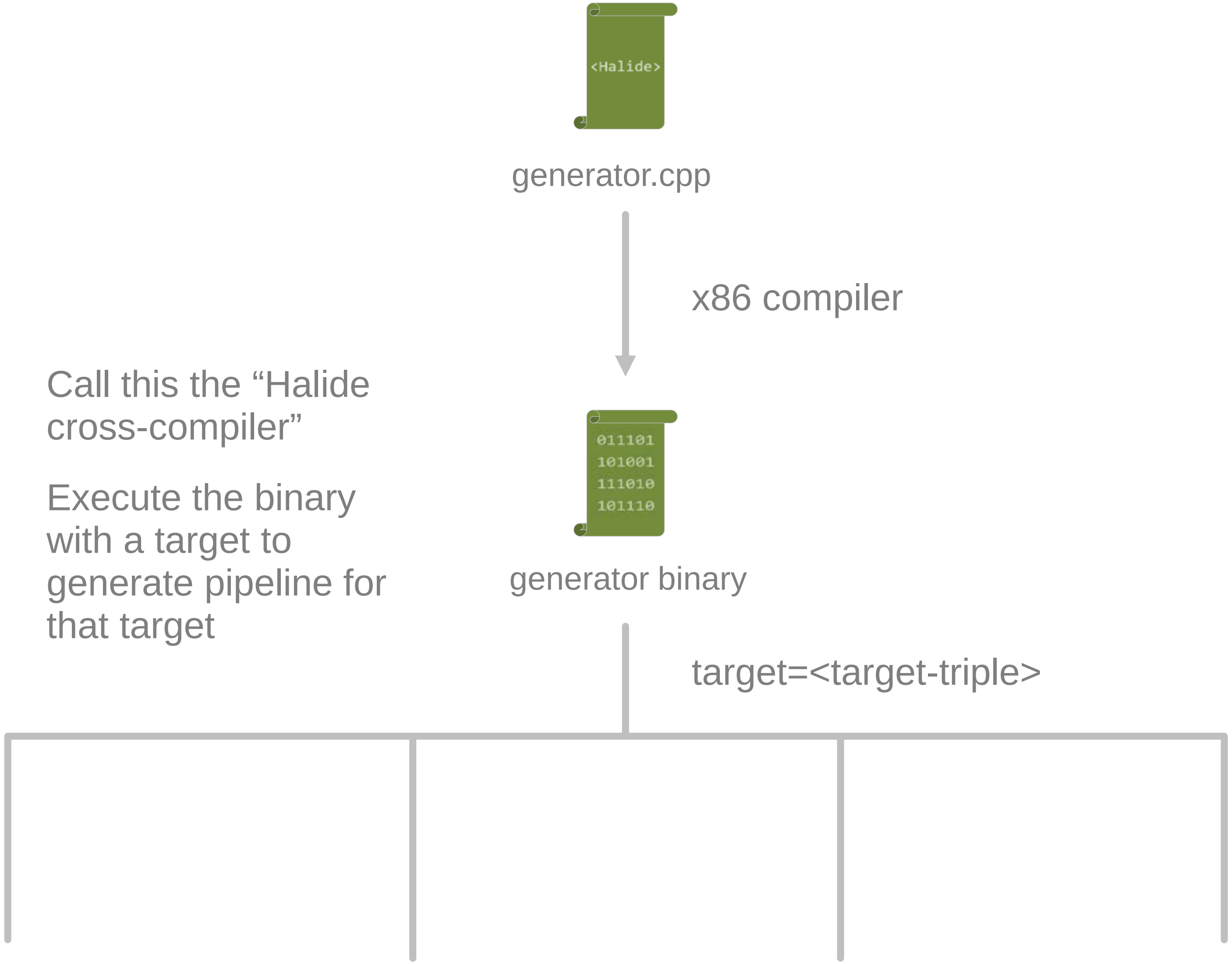
- Compile Halide code to x86 binary using g++/clang++. Link with libHalide.a or libHalide.so
- The generated binary file can emit code for different targets
- Target is specified as a triple: `<host>-<bits-<os>-<features>`

| Host | arm, x86, hexagon |
|---|---|
| Bits | 32, 64 |
| Os | android, qurt, noos |
| Features | hvx_64, hvx_128, hvx_v62 |



Call this the "Halide cross-compiler"

Execute the binary with a target to generate pipeline for that target



generator.cpp

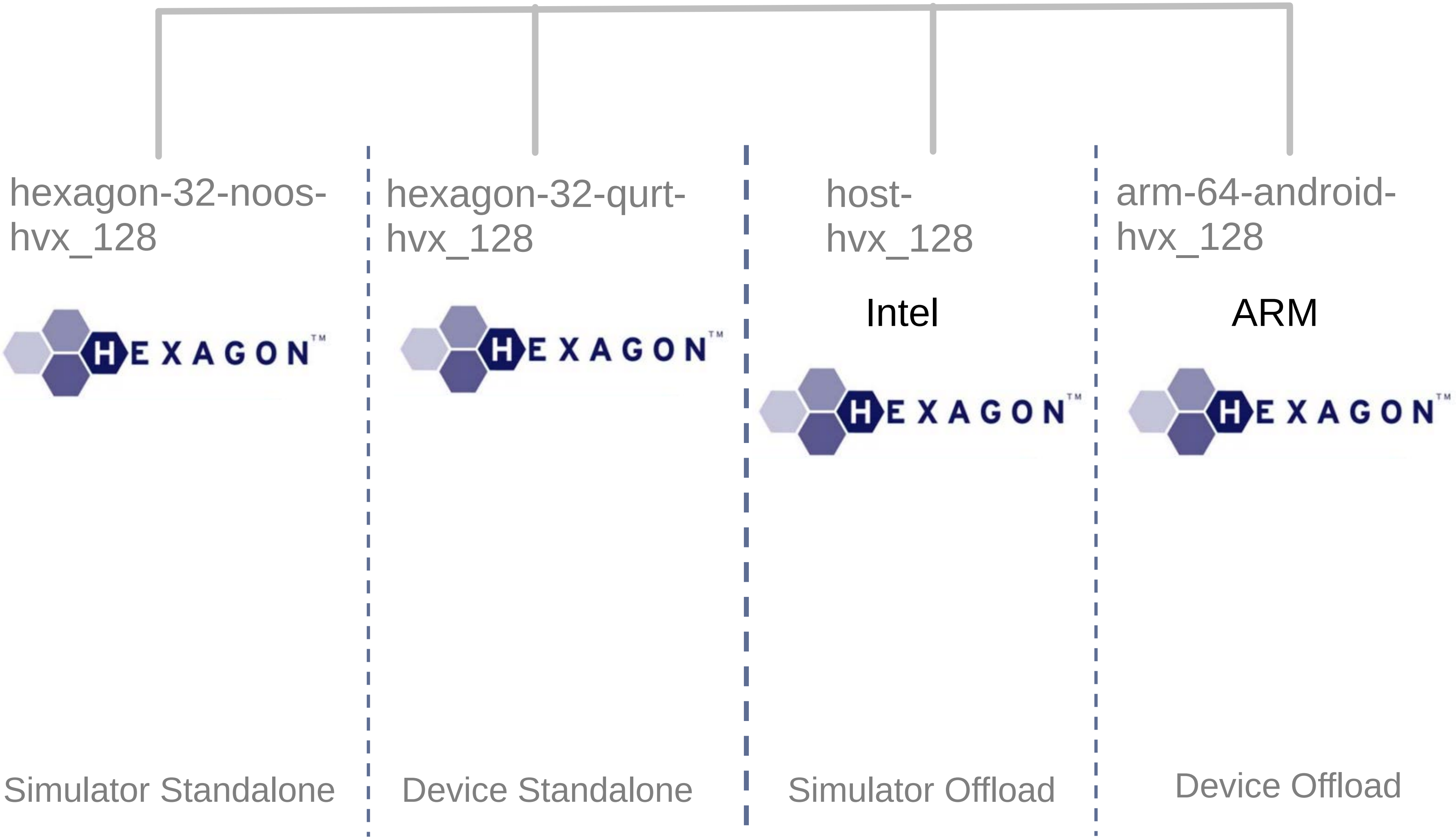
x86 compiler

generator binary

target=<target-triple>

Call this the “Halide cross-compiler”

Execute the binary with a target to generate pipeline for that target



hexagon-32-noos-hvx_128

hexagon-32-qurt-hvx_128

host-hvx_128

arm-64-android-hvx_128

Intel

ARM

Simulator Standalone

Device Standalone

Simulator Offload

Device Offload

# Offload Execution Model – Heterogenous computing model

- Some or all stages of a pipeline can be executed on HVX
- Offloading is easy to realize in Halide code and is transparent – the underlying details of data transfer between the CPU and the Hexagon DSP are hidden from the Halide programmer
- Device-Offload mode
  - This is used for on-target execution where the pipeline is run on actual Hexagon hardware (CPU is ARM)
- Simulator-Offload mode
  - This is used when the part of the pipeline scheduled to run on Hexagon is simulated using the Hexagon Instruction Set Simulator (CPU is X86)
  - Can be used without any Hexagon hardware

# Offload Execution Model – Device-Offload vs. Simulator-Offload

| Device-Offload mode | Simulator-Offload mode |
|---|---|
| Offloading is achieved by adding ".hexagon()" directive in the Halide program | Offloading is achieved by adding ".hexagon()" directive in the Halide program |
| The host is the ARM CPU and the Hexagon side is not simulated but executed on actual hardware | The host is x86 whereas the Hexagon side is simulated using the Hexagon instruction set simulator |
| Functions in the runtime use QuRT™ software to do things such as acquire and release HVX contexts, thread pool management etc. | Functions in the runtime that require QuRT are implemented using a shim library |
| Model developed for MTP and Dragonboard environments | This model can be used when working on an x86 based desktop with the Hexagon SDK installed |
| Communication between the host and the device (Hexagon) is done using FastRPC, under the hood. | Communication between the host and the device (Hexagon) is done using a message passing interface. |

# Offload Execution Model – Device-Offload vs. Simulator-Offload (cont.)

| Device-Offload mode | Simulator-Offload mode |
|---|---|
| Offloading is achieved by a program | )” directive in the Halide |
| The host is the ARM CPU a executed on actual hardwa | s simulated using the |
| Functions in the runtime us acquire and release HVX c | are implemented using a |
| Model developed for MTP a | n x86 based desktop with |
| Communication between th using FastRPC, under the l | evice (Hexagon) is done |

There is **no change** needed to your application or your Halide pipeline to switch between Device-Offload and Simulator-Offload modes. Let’s see an example now

# Offload Execution Model – Sample Offloaded Halide Pipeline

```
class Dilate3x3 : public Generator<Dilate3x3> {
public:
    // Takes an 8 bit image; one channel.
    Input<Buffer<uint8_t>> input{"input", 2};
    // Outputs an 8 bit image; one channel.
    Output<Buffer<uint8_t>> output{"output", 2};

    void generate() {
     bounded_input(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
     max_y(x, y) = max(bounded_input(x, y-1), bounded_input(x, y),
                       bounded_input(x, y+1));
     output(x, y) = max(max_y(x-1, y), max_y(x, y), max_y(x+1, y));
    }
    //Schedule
};
```

A generator is a structured way of doing AOT compilation in Halide

# Offload Execution Model – Sample Offloaded Halide Pipeline (cont.)

```
class Dilate3x3 : public Generator<Dilate3x3> {
public:
    // Takes an 8 bit image; one channel.
    Input<Buffer<uint8_t>> input{"input", 2};
    // Outputs an 8 bit image; one channel.
    Output<Buffer<uint8_t>> output{"output", 2};

    void generate() {
     bounded_input(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
     max_y(x, y) = max(bounded_input(x, y-1), bounded_input(x, y),
                       bounded_input(x, y+1));
     output(x, y) = max(max_y(x-1, y), max_y(x, y), max_y(x+1, y));
    }
    //Schedule
};
```

Declare the parameters to the pipeline as member variables. These appear in the signature of the generated function

# Offload Execution Model – Sample Offloaded Halide Pipeline (cont.)

```
class Dilate3x3 : public Generator<Dilate3x3> {
public:
    // Takes an 8 bit image; one channel.
    Input<Buffer<uint8_t>> input{"input", 2};
    // Outputs an 8 bit image; one channel.
    Output<Buffer<uint8_t>> output{"output", 2};

    void generate() {
     bounded_input(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
     max_y(x, y) = max(bounded_input(x, y-1), bounded_input(x, y),
                       bounded_input(x, y+1));
     output(x, y) = max(max_y(x-1, y), max_y(x, y), max_y(x+1, y));
    }
    //Schedule
};
```

Define the generate member function to define the “algorithm”. This algorithm computes the max value of a 3x3 block of pixels

# Offload Execution Model – Sample Offloaded Halide Pipeline (cont.)

```
class Dilate3x3 : public Generator<Dilate3x3> {
//Schedule
 void schedule() {
   Var xi{"xi"}, yi{"yi"};
   int vector_size = 128;
   bounded_input
           .compute_at(Func(output), y)
           .align_storage(x, 128)
           .vectorize(x, vector_size, TailStrategy::RoundUp);
   output
         .hexagon()
         .tile(x, y, xi, yi, vector_size, 4)
         .vectorize(xi)
         .unroll(yi);
  }
};
```

Define an appropriate schedule for organizing the computation of your algorithm. Iterate over the schedule to progressively improve performance

# Offload Execution Model – How to Build and Run

- Assuming the host for the build is x86, then using the x86 compiler, compile the generator into an executable

```
${x86_CXX} -std=c++11 –I ${INCLUDE_DIRS} ${CFLAGS} ${HALIDE_PATH}/tools/GenGen.cpp your_generator.cpp
${HALIDE_PATH}/lib/libHalide.a –o your_generator –ldl –lpthread -lz
```

- Run the generator to produce an object file (“.o”) and a header file (“.h”) containing the signature of your pipeline

```
./your_generator –o . –e o,h –f pipeline target=<desired_target>
```

Desired output. Comma separated values from “o, h, assembly, bitcode”

Device-Offload mode:
`arm-64-android-hvx_128`
`(or hvx_64)`
Simulator-offload mode:
`host-hvx_128 (or hvx_64)`

```
$> ls .
pipeline.h pipeline.o
```

# Offload Execution Model – Sample Application Code (Calls Halide Pipeline)

```
#include “pipeline.h”
int main(int argc, char **argv) {
 Halide::Runtime::Buffer<uint8_t> in(nullptr, 1024, 1024);
 Halide::Runtime::Buffer<uint8_t> out(nullptr, 1024, 1024);

 in.device_malloc(halide_hexagon_device_interface());
 out.device_malloc(halide_hexagon_device_interface());

 in.for_each_value([&](uint8_t &x) {
     x = static_cast<uint8_t>(rand());
 });

 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_turbo);
 halide_hexagon_power_hvx_on(NULL);

 int result = pipeline(in, out);

 halide_hexagon_power_hvx_off(NULL);
 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_default);

 printf("Success!\n");
}
```

# Offload Execution Model – Sample Application Code (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
 Halide::Runtime::Buffer<uint8_t> in(nullptr, 1024, 1024);
 Halide::Runtime::Buffer<uint8_t> out(nullptr, 1024, 1024);

 in.device_malloc(halide_hexagon_device_interface());
 out.device_malloc(halide_hexagon_device_interface());

 in.for_each_value([&](uint8_t &x) {
     x = static_cast<uint8_t>(rand());
 });

 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_turbo);
 halide_hexagon_power_hvx_on(NULL);

 int result = pipeline(in, out);

 halide_hexagon_power_hvx_off(NULL);
 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_default);

 printf("Success!\n");
}
```

Header file containing the signature of the entry point of the pipeline. Created by Halide

# Offload Execution Model – Sample Application Code (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
 Halide::Runtime::Buffer<uint8_t> in(nullptr, 1024, 1024);
 Halide::Runtime::Buffer<uint8_t> out(nullptr, 1024, 1024);

 in.device_malloc(halide_hexagon_device_interface());
 out.device_malloc(halide_hexagon_device_interface());

 in.for_each_value([&](uint8_t &x) {
     x = static_cast<uint8_t>(rand());
 });

 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_turbo);
 halide_hexagon_power_hvx_on(NULL);

 int result = pipeline(in, out);

 halide_hexagon_power_hvx_off(NULL);
 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_default);

 printf("Success!\n");
}
```

Define 1024x1024 8-bit buffers (input and output of pipeline). “nullptr” indicates no memory allocated on the host side for the buffers

# Offload Execution Model – Sample Application Code (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
 Halide::Runtime::Buffer<uint8_t> in(nullptr, 1024, 1024);
 Halide::Runtime::Buffer<uint8_t> out(nullptr, 1024, 1024);

 in.device_malloc(halide_hexagon_device_interface());
 out.device_malloc(halide_hexagon_device_interface());

 in.for_each_value([&](uint8_t &x) {
     x = static_cast<uint8_t>(rand());
 });

 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_turbo);
 halide_hexagon_power_hvx_on(NULL);

 int result = pipeline(in, out);

 halide_hexagon_power_hvx_off(NULL);
 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_default);

 printf("Success!\n");
}
```

Use the device interface for hexagon to allocate memory for the buffers on the device (Hexagon). Since host pointer is null, it’ll use the ion allocator to allocate shared memory between the host and the Hexagon DSP so that we have zero-copy buffers for data transfer. Initialize buffers

# Offload Execution Model – Sample Application Code (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
 Halide::Runtime::Buffer<uint8_t> in(nullptr, 1024, 1024);
 Halide::Runtime::Buffer<uint8_t> out(nullptr, 1024, 1024);

 in.device_malloc(halide_hexagon_device_interface());
 out.device_malloc(halide_hexagon_device_interface());

 in.for_each_value([&](uint8_t &x) {
     x = static_cast<uint8_t>(rand());
 });

 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_turbo);
 halide_hexagon_power_hvx_on(NULL);

 int result = pipeline(in, out);

 halide_hexagon_power_hvx_off(NULL);
 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_default);

 printf("Success!\n");
}
```

Set Hexagon in performance mode and power HVX on

# Offload Execution Model – Sample Application Code (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
 Halide::Runtime::Buffer<uint8_t> in(nullptr, 1024, 1024);
 Halide::Runtime::Buffer<uint8_t> out(nullptr, 1024, 1024);

 in.device_malloc(halide_hexagon_device_interface());
 out.device_malloc(halide_hexagon_device_interface());

 in.for_each_value([&](uint8_t &x) {
     x = static_cast<uint8_t>(rand());
 });

 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_turbo);
 halide_hexagon_power_hvx_on(NULL);

 int result = pipeline(in, out);

 halide_hexagon_power_hvx_off(NULL);
 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_default);

 printf("Success!\n");
}
```

Invoke the pipeline!

# Offload Execution Model – Sample Application Code (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
 Halide::Runtime::Buffer<uint8_t> in(nullptr, 1024, 1024);
 Halide::Runtime::Buffer<uint8_t> out(nullptr, 1024, 1024);

 in.device_malloc(halide_hexagon_device_interface());
 out.device_malloc(halide_hexagon_device_interface());

 in.for_each_value([&](uint8_t &x) {
     x = static_cast<uint8_t>(rand());
 });

 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_turbo);
 halide_hexagon_power_hvx_on(NULL);

 int result = pipeline(in, out);

 halide_hexagon_power_hvx_off(NULL);
 halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_default);

 printf("Success!\n");
}
```

Turn HVX off and reset power mode

# Offload Execution Model – How to Build and Run

- Assuming the host for the build is x86, then using the x86 compiler, compile the generator into an executable

```
${x86_CXX} -std=c++11 –I ${INCLUDE_DIRS} ${CFLAGS} ${HALIDE_PATH}/tools/GenGen.cpp your_generator.cpp
${HALIDE_PATH}/lib/libHalide.a –o your_generator –ldl –lpthread -lz
```

- Run the generator to produce an object file (“.o”) and a header file (“.h”) containing the signature of your pipeline

```
./your_generator –o . –e o,h –f pipeline target=<desired_target>
```

- Include the header created in the previous step into your application file. Compile the application file and link in the object file for your pipeline created in the previous step

Device-Offload mode:
ARM compiler
Simulator-Offload mode:
x86 Compiler

```
${CXX} application.cpp pipeline.o –o your_application
```

# Standalone Execution Model

- All  the stages of the pipeline are executed on Hexagon (HVX)
- Data transfer between the host CPU and Hexagon DSP are the programmers responsibility
- Device-Standalone mode
  - Used for on-target execution where the pipeline is run on actual Hexagon hardware (CPU is ARM)
- Simulator-Standalone mode
  - Used when the pipeline is simulated on “hexagon-sim”, the Hexagon Simulator. The application file is also compiled for Hexagon and executed on “hexagon-sim”

# Standalone Execution Model – Device-Standalone vs. Simulator-Standalone

| Device-Standalone mode | Simulator-Standalone mode |
|---|---|
| ~~Offloading is achieved by adding “`.hexagon()`” directive in the Halide program~~ | ~~Offloading is achieved by adding “`.hexagon()`” directive in the Halide program~~ |
| The host is the ARM CPU and the Hexagon side is not simulated but executed on actual hardware | The host and device are both Hexagon and both are simulated |
| Functions in the runtime use QuRT to do things such as acquire and release HVX contexts, thread pool management etc | Functions in the runtime use the Standalone library that is shipped as part of the Hexagon Tools |
| Model developed for MTP and Dragonboard environments | This model can be used when working on an x86 based desktop with the Hexagon SDK installed |
| Communication between the host and the device (Hexagon) is the responsibility of the programmer | Communication between the host and the device (Hexagon) is the responsibility of the programmer |

# Standalone Execution Model – Device-Standalone vs. Simulator-Standalone (cont.)

| Device-Standalone mode | Simulator-Standalone mode |
|---|---|
| ~~Offloading is achieved by adding “.hexagon()” directive in the Halide program~~ | ~~Offloading is achieved by adding “.hexagon()” directive in the Halide program~~ |
| The host is the ARM CPU executed on actual hardw | both are simulated |
| Functions in the runtime u release HVX contexts, thr | e library that is shipped |
| Model developed for MTP | n x86 based desktop with |
| Communication between t responsibility of the progra | device (Hexagon) is the |

There is **no change** needed to ~~your application or~~ your Halide pipeline to switch between Device-standalone and Simulator-standalone modes. Let’s see an example now

# Standalone Execution Model – Sample Standalone Halide Pipeline

```
class Dilate3x3 : public Generator<Dilate3x3> {
public:
    // Takes an 8 bit image; one channel.
    Input<Buffer<uint8_t>> input{"input", 2};
    // Outputs an 8 bit image; one channel.
    Output<Buffer<uint8_t>> output{"output", 2};

    void generate() {
     bounded_input(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
     max_y(x, y) = max(bounded_input(x, y-1), bounded_input(x, y),
                       bounded_input(x, y+1));
     output(x, y) = max(max_y(x-1, y), max_y(x, y), max_y(x+1, y));
    }
    //Schedule
};
```

A generator is a structured way of doing AOT compilation in Halide

# Standalone Execution Model – Sample Standalone Halide Pipeline (cont.)

```
class Dilate3x3 : public Generator<Dilate3x3> {
public:
    // Takes an 8 bit image; one channel.
    Input<Buffer<uint8_t>> input{"input", 2};
    // Outputs an 8 bit image; one channel.
    Output<Buffer<uint8_t>> output{"output", 2};

    void generate() {
     bounded_input(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
     max_y(x, y) = max(bounded_input(x, y-1), bounded_input(x, y),
                       bounded_input(x, y+1));
     output(x, y) = max(max_y(x-1, y), max_y(x, y), max_y(x+1, y));
    }
    //Schedule
};
```

Declare the parameters to the pipeline as member variables. These appear in the signature of the generated function

# Standalone Execution Model – Sample Standalone Halide Pipeline (cont.)

```
class Dilate3x3 : public Generator<Dilate3x3> {
public:
    // Takes an 8 bit image; one channel.
    Input<Buffer<uint8_t>> input{"input", 2};
    // Outputs an 8 bit image; one channel.
    Output<Buffer<uint8_t>> output{"output", 2

    void generate() {
     bounded_input(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
     max_y(x, y) = max(bounded_input(x, y-1), bounded_input(x, y),
                       bounded_input(x, y+1));
     output(x, y) = max(max_y(x-1, y), max_y(x, y), max_y(x+1, y));
    }
    //Schedule
};
```

Define the generate member function to define the “algorithm”. This algorithm computes the max value of a 3x3 block of pixels

# Standalone Execution Model – Sample Standalone Halide Pipeline (cont.)

```
class Dilate3x3 : public Generator<Dilate3x3> {
//Schedule
 void schedule() {
   Var xi{"xi"}, yi{"yi"};
   int vector_size = 128;
   bounded_input
           .compute_at(Func(output), y)
           .align_storage(x, 128)
           .vectorize(x, vector_size, TailStrat
   output
         ~~.hexagon()~~
         .tile(x, y, xi, yi, vector_size, 4)
         .vectorize(xi)
         .unroll(yi);
  }
};
```

Define an appropriate schedule for organizing the computation of your algorithm. Iterate over the schedule to progressively improve performance.
**“.hexagon()” directive not needed**

# Standalone Execution Model – How to Build and Run

- Assuming the host for the build is x86, then using the x86 compiler, compile the generator into an executable

```
${x86_CXX} -std=c++11 –I ${INCLUDE_DIRS} ${CFLAGS} ${HALIDE_PATH}/tools/GenGen.cpp
your_generator.cpp ${HALIDE_PATH}/lib/libHalide.a –o your_generator –ldl –lpthread -lz
```

- Run the generator to produce an object (“.o”) file and a header file (“.h”) containing the signature of your pipeline

```
./your_generator –o . –e o,h –f pipeline target=<desired_target>
```

Desired output. Comma separated values from “o, h, assembly, bitcode”

Device-Standalone mode:
`hexagon-32-`**`qurt`**`-hvx_128`
`(or hvx_64)`
Simulator-Standalone mode:
`hexagon-32-`**`noos`**`-hvx_128`
`(or hvx_64)`

```
$> ls .
pipeline.h pipeline.o
```

# Standalone Execution Model – Sample Application Code  (Calls Halide Pipeline)

```
#include “pipeline.h”
int main(int argc, char **argv) {
int width  = atoi(argv[1]);
int height = atoi(argv[2]);
unsigned char *input  = (unsigned char *)memalign(1 << LOG2VLEN, width*height*sizeof(unsigned char));
unsigned char *output = (unsigned char *)memalign(1 << LOG2VLEN, width*height*sizeof(unsigned char));

 /* initialize input */
 for (int i = 0; i < height, ++i) {
  for (int j = 0; j < width; ++j) {
   input[i*width + j] = rand() & 0xFF;
  }
 }
 halide_dimension_t x_dim{0, width, 1};
 halide_dimension_t y_dim{0, height, width};
 halide_dimension_t io_shape[2] = {x_dim, y_dim};
 Halide::Runtime::Buffer<uint8_t> input_buf(input, dims, io_shape);
 Halide::Runtime::Buffer<uint8_t> output_buf(output, dims, io_shape);

 int result = pipeline(input_buf, output_buf);

 printf("Success!\n");
}
```

# Standalone Execution Model – Sample Application Code  (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
int width  = atoi(argv[1]);
int height = atoi(argv[2]);
unsigned char *input  = (unsigned char *)memalign(1 << LOG2VLEN, w
unsigned char *output = (unsigned char *)memalign(1 << LOG2VLEN, w

 /* initialize input */
 for (int i = 0; i < height, ++i) {
  for (int j = 0; j < width; ++j) {
   input[i*width + j] = rand() & 0xFF;
  }
 }
 halide_dimension_t x_dim{0, width, 1};
 halide_dimension_t y_dim{0, height, width};
 halide_dimension_t io_shape[2] = {x_dim, y_dim};
 Halide::Runtime::Buffer<uint8_t> input_buf(input, dims, io_shape);
 Halide::Runtime::Buffer<uint8_t> output_buf(output, dims, io_shape);

 int result = pipeline(input_buf, output_buf);

 printf("Success!\n");
}
```

Header file containing the signature of the entry point of the pipeline. Created by Halide

# Standalone Execution Model – Sample Application Code  (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
int width  = atoi(argv[1]);
int height = atoi(argv[2]);
unsigned char *input  = (unsigned char *)memalign(1 << LOG2VLEN, width*height*sizeof(unsigned char));
unsigned char *output = (unsigned char *)memalign(1 << LOG2VLEN, width*height*sizeof(unsigned char));

 /* initialize input */
 for (int i = 0; i < height, ++i) {
  for (int j = 0; j < width; ++j) {
   input[i*width + j] = rand() & 0xFF;
  }
 }
 halide_dimension_t x_dim{0, wid
 halide_dimension_t y_di
 halide_dimension_t i
 Halide::Runtime::Bu                       dims, io_shape
 Halide::Runtime::                           dims, io_sha

 int result = pipe

 printf("Success!\n"
}
```

This is Simulator-standalone mode. For Device-standalone, we recommend using the “`rpcmem`” library

Define aligned input and output buffers and initialize the input

# Standalone Execution Model – Sample Application Code  (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
int width  = atoi(argv[1]);
int height = atoi(argv[2]);
unsigned char *input  = (unsigned char *)memalign(1 << LOG2VLEN, width*height*sizeof(unsigned char));
unsigned char *output = (unsigned char *)memalign(1 << LOG2VLEN, width*height*sizeof(unsigned char));

 /* initialize input */
 for (int i = 0; i < height, ++i) {
  for (int j = 0; j < width; ++j) {
   input[i*width + j] = rand() & 0xFF;
  }
 }
 halide_dimension_t x_dim{0, width, 1};
 halide_dimension_t y_dim{0, height, width};
 halide_dimension_t io_shape[2] = {x_dim, y_dim};
 Halide::Runtime::Buffer<uint8_t> input_buf(input, dims, io_shape);
 Halide::Runtime::Buffer<uint8_t> output_buf(output, dims, io_shape);

 int result = pipeline(input_buf, output_buf);

 printf("Success!\n");
}
```

Set up Halide buffers

# Standalone Execution Model – Sample Application Code  (Calls Halide Pipeline) (cont.)

```
#include “pipeline.h”
int main(int argc, char **argv) {
int width  = atoi(argv[1]);
int height = atoi(argv[2]);
unsigned char *input  = (unsigned char *)memalign(1 << LOG2VLEN, width*height*sizeof(unsigned char));
unsigned char *output = (unsigned char *)memalign(1 << LOG2VLEN, width*height*sizeof(unsigned char));

 /* initialize input */
 for (int i = 0; i < height, ++i) {
  for (int j = 0; j < width; ++j) {
   input[i*width + j] = rand() & 0xFF;
  }
 }
 halide_dimension_t x_dim{0, width, 1};
 halide_dimension_t y_dim{0, height, width};
 halide_dimension_t io_shape[2] = {x_dim, y_dim};
 Halide::Runtime::Buffer<uint8_t> input_buf(input, dims, io_shape
 Halide::Runtime::Buffer<uint8_t> output_buf(output, dims, io_sha

 int result = pipeline(input_buf, output_buf);

 printf("Success!\n");
}
```

Invoke the pipeline!

# Standalone Execution Model – How to Build and Run

- Assuming the host for the build is x86, then using the x86 compiler, compile the generator into an executable

```
${x86_CXX} -std=c++11 –I ${INCLUDE_DIRS} ${CFLAGS} ${HALIDE_PATH}/tools/GenGen.cpp
your_generator.cpp ${HALIDE_PATH}/lib/libHalide.a –o your_generator –ldl –lpthread -lz
```

- Run the generator to produce an object (“.o”) file and a header file (“.h”) containing the signature of your pipeline

```
./your_generator –o . –e o,h –f pipeline target=<desired_target>
```

- Include the header created in the previous step into your application file. Compile the application file and link in the object file for your pipeline created in the previous step

Simulator-Standalone mode:
`hexagon-clang++`

```
${CXX} application.cpp your_pipeline.o –o your_application
```

# Part 4 – Performance Optimizations

1

Introduction to Halide

2

The Halide Programming Language

3

Halide for HVX

4

Performance Optimizations

5

Installation Instructions and Troubleshooting

# Profiling – Using the Halide Profiler

- The Halide profiler enables the programmer to measure time spent in stages that haven’t been inlined
- Used in Device-Offload or Simulator-Offload mode

Sample profile for dilate3x3

```
04-08 08:13:10.217 14319 14319 I halide  : dilate3x3_hvx128
04-08 08:13:10.217 14319 14319 I halide  :  total time: 190.178009 ms  samples: 903  runs: 100  time/run:
1.901780 ms
04-08 08:13:10.217 14319 14319 I halide  :  average threads used: 0.775194
04-08 08:13:10.217 14319 14319 I halide  :  heap allocations: 0  peak heap usage: 0 bytes
04-08 08:13:10.217 14319 14319 I halide  :   output:                0.802ms   (42%)   threads: 0.442
04-08 08:13:10.217 14319 14319 I halide  :   bounded_input:         1.099ms   (57%)   threads: 1.000
```

# Profiling – Using the Halide Profiler (cont.)

Use the following steps to get profile output.

- Add the “`profile`” feature to the target while running your generator.

```
./your_generator –o . –e o,h –f pipeline target=arm-64-android-hvx_128-profile
```

- Add a call to “`halide_profiler_report(nullptr)`” to the end of your application file.
- Rebuild your application.
- If running on a device, run “`adb logcat –s halide`” while running your application to view the generated profile data.

# Profiling – Using the Halide Profiler (cont.)

```
04-08 08:13:10.217 14319 14319 I halide  : dilate3x3_hvx128
04-08 08:13:10.217 14319 14319 I halide  :  total time: 190.178009 ms  samples: 903  runs: 100
time/run: 1.901780 ms
04-08 08:13:10.217 14319 14319 I halide  :  average threads used: 0.775194
04-08 08:13:10.217 14319 14319 I halide  :  heap allocations: 0  peak heap usage: 0 bytes
04-08 08:13:10.217 14319 14319 I halide  :   output:               0.802ms   (42%)   threads: 0.442
04-08 08:13:10.217 14319 14319 I halide  :   bounded_input:        1.099ms   (57%)   threads: 1.000
```

Shows stages with schedule. Inline stages do not show up in profile

# Terminology

- Application File – Code that calls the Halide pipeline
- External buffers – Memory buffers external to the pipeline that are either inputs to or outputs of the pipeline. These are allocated and set up in the application file
- Intermediate / Internal storage – Storage for a Func that has not been scheduled inline and is not an input buffer or an output Func

```
producer(x) = in(x) + 2;
consumer(x) = producer(x-1) + producer(x);
consumer.vectorize(x, 128);
producer.compute_at(consumer, y).vectorize (x, 128);
```

Memory needs to be allocated to store “producer”. This is called internal storage. This memory is not managed by the programmer. “producer” is also called an Intermediate Func

- Inline schedule – The default schedule for a Func when no schedule is specified

```
producer(x) = in(x) + 2;
consumer(x) = producer(x-1) + producer(x);
consumer.vectorize (x, 128);
```

No schedule for producer, so it is computed inline inside consumer. The consumer is effectively: `consumer(x) = in(x-1) + 2 + in(x) + 2;`

# Terminology – halide_buffer_t

- This is the data structure used to represent an image in generated Halide code
- If a pipeline has one image input and one image output then it's signature will contain pointers to two halide_buffer_t objects for one input image and one output image
- A halide_buffer_t structure has the following information about the image
  - Pointer to the data of the image in memory
  - Type of each image element
  - Number of dimensions of the image
  - Information about each dimension
    - min – the smallest coordinate of the dimension; default is 0
    - extent – the size of the dimension
    - stride – The number of elements between consecutive elements of this dimension

# Setting Up halide_buffer_t for External Buffers

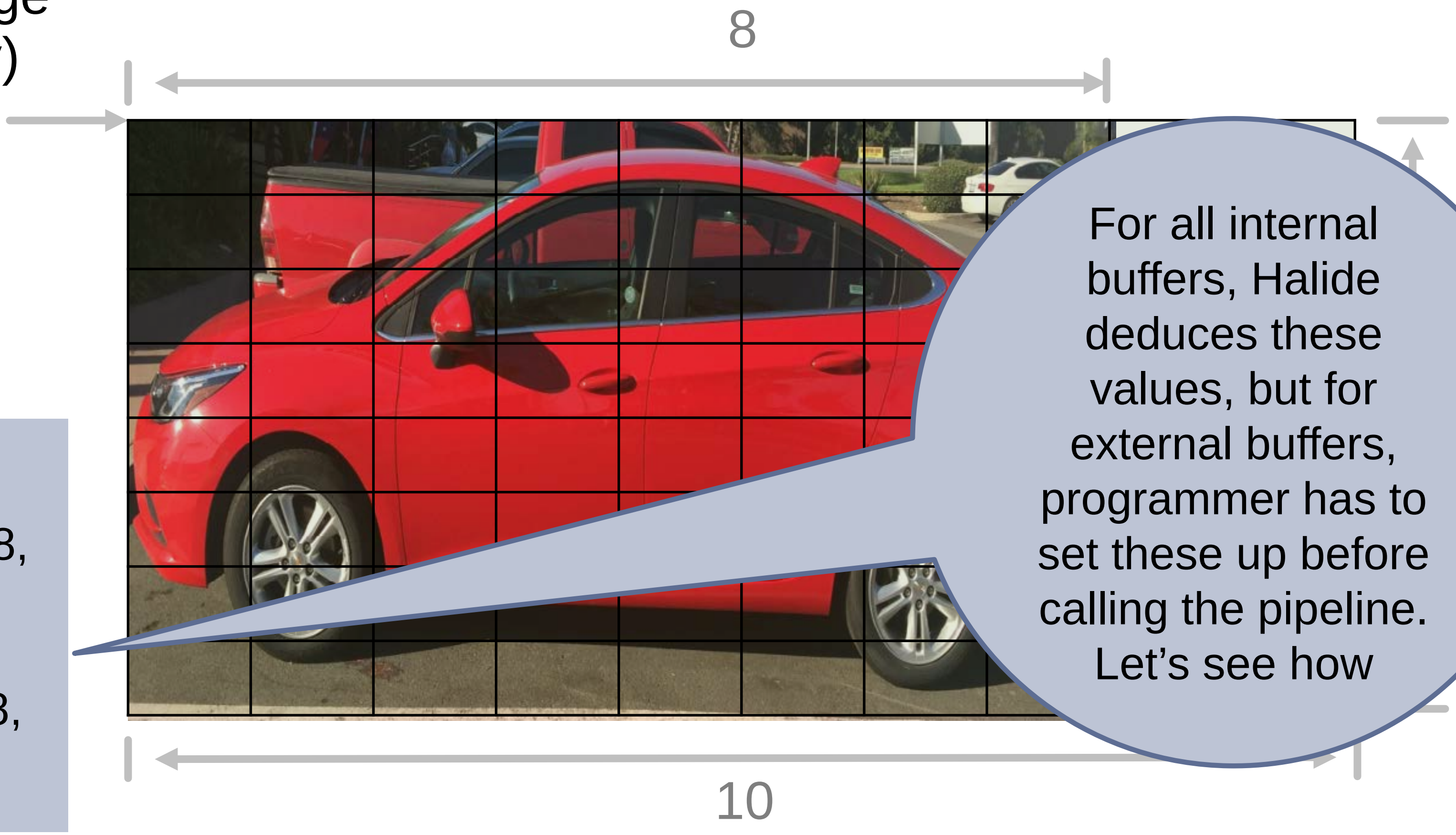
```
const int vlen   = get_target().has_feature(Target::HVX_128) ? 128 : 64;
const int stride = (width + vlen-1)&(-vlen);  // Align rows to vector length

halide_buffer_t input_buf = {0};
halide_dimension_t in_dim[2] = {0};
halide_dimension_t out_dim[2] = {0};

input_buf.type.code = halide_type_uint;
input_buf.type.bits = 8;             // Element size in bits
input_buf.type.lanes = 1;
input_buf.dimensions = 2;

input_buf.dim = in_dim;
input_buf.dim[0].extent = width;        // Image dimensions
input_buf.dim[1].extent = height;
input_buf.dim[0].stride = 1;         // Stride for each dimension
input_buf.dim[1].stride = stride;
input_buf.dim[0].min = 0;            // Index of upper left element
input_buf.dim[1].min = 0;
```

Setting up halide_buffer_t for a “width” x “height” 2 dimensional image

Good for performance if image width is not a multiple of vector size. Leads to aligned loads and stores. Allocate memory for buffer accordingly

# halide_buffer_t Example

host (ptr to image data in memory)



**‘x’ dimension**
min = 0, extent = 8,
stride = 1
**‘y’ dimension**
min = 0, extent = 8,
stride = 10

For all internal buffers, Halide deduces these values, but for external buffers, programmer has to set these up before calling the pipeline. Let’s see how

# Halide::Runtime::Buffer

- Wrapper for halide_buffer_t that simplifies code

```
const int vlen   = get_target().has_feature(Target::HVX_128) ? 128 : 64;
const int stride = (width + vlen-1)&(-vlen);  // Align rows to vector lengt

halide_buffer_t input_buf = {0};
halide_d
halide_d

input_bu
input_bu
input_bu
input_bu

input_bu
input_buf.dim[0].extent = width;         // Image dimensions
input_buf.dim[1].extent = height;
input_buf.dim[0].stride = 1;          // Stride for each dimension
input_buf.dim[1].stride = stride;
input_buf.dim[0].min = 0;             // Index of upper left element
input_buf.dim[1].min = 0;
```

Setting up halide_buffer_t for a “width” x “height” 2 dimensional image

```
Halide::Runtime::Buffer<uint8_t> in(nullptr, width height);
Halide::Runtime::Buffer<uint8_t> out(nullptr, width, height);
in.device_malloc(halide_hexagon_device_interface());
out.device_malloc(halide_hexagon_device_interface());
```

ance if multiple ads to aligned loads and stores. Allocate memory for buffer accordingly

# Performance Considerations – Memory Allocation and Zero-Copy Buffers

- A zero-copy buffer is memory that is visible to the host processor and the Hexagon processor. When working with zero-copy buffers, Halide knows that it does not need to perform a copy when switching access between the host and the Hexagon processor.
- When writing Halide applications, it is recommended that programmers use halide_device_malloc and halide_device_free to manage zero-copy memory

Use in Device-Offload and Simulator-Offload mode

```
#include "HalideRuntimeHexagonHost.h"
...
// Allocate buffers
halide_device_malloc(nullptr, &input_buf,  halide_hexagon_device_interface());
if (input_buf.host == NULL) {
  return 1;
}
...
// Free buffers
halide_device_free(NULL, &input_buf);
```

# Performance Considerations – Memory Allocation and Zero-Copy Buffers (cont.)

- If you are adding Halide to an existing application that uses the SDK rpcmem library for allocating zero copy buffers already, then you need to attach such memory to a halide_buffer_t object using halide_hexagon_wrap_device_handle

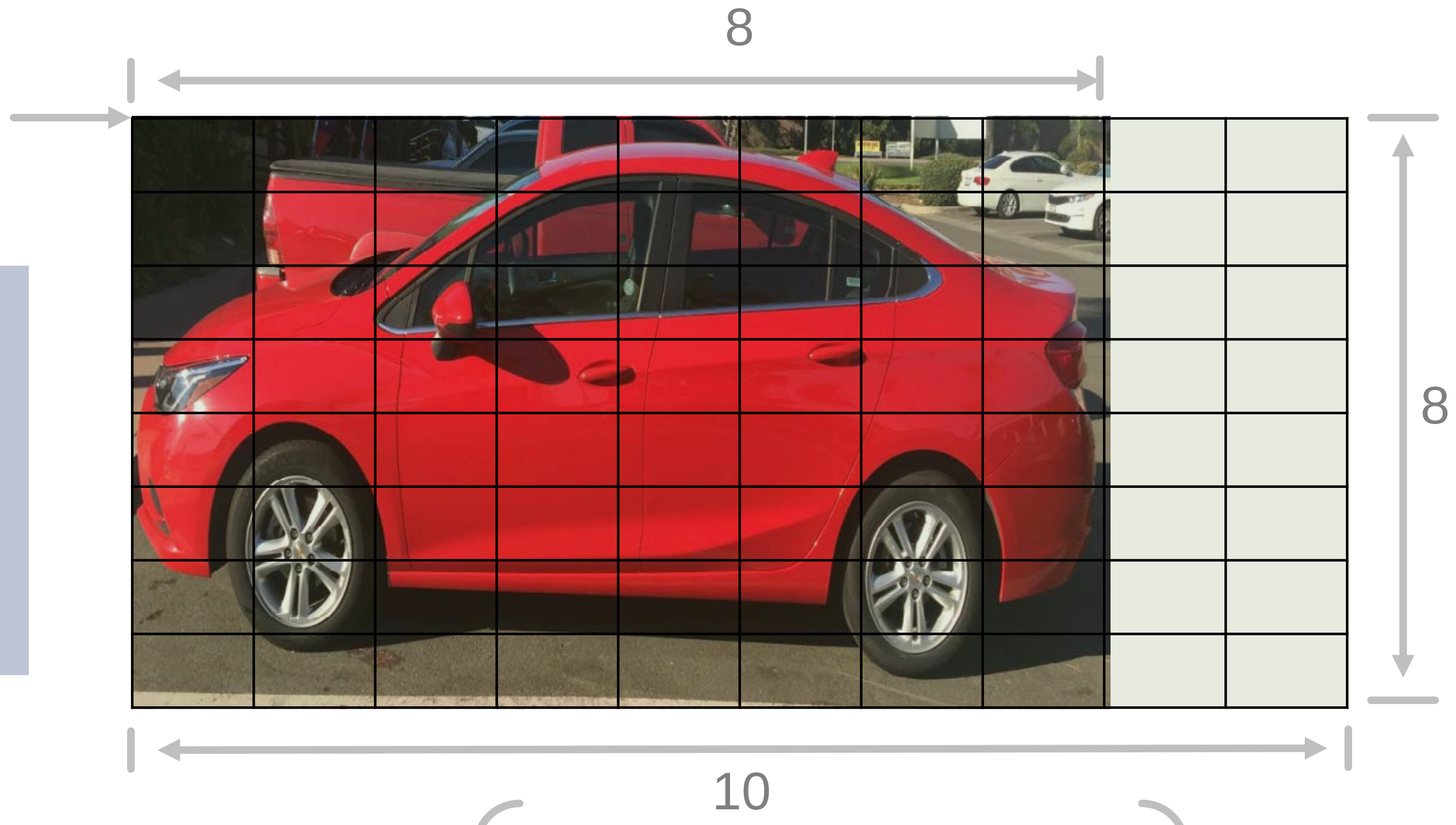
Use in Device-Standalone mode

```
#include "rpcmem.h"
#include "HalideRuntimeHexagonHost.h"
...
rpcmem_init(0);
...
// Allocate buffers
const int bufsize = stride * height + VLEN;  // Over-allocate by one vector
input_buf.host  = (uint8_t*)rpcmem_alloc(25, RPCMEM_DEFAULT_FLAGS, bufsize);
if (input_buf.host == NULL) {
   return 1;
}
halide_hexagon_wrap_device_handle(nullptr, &input_buf, input_buf.host, bufsize);
...
// Free buffers
rpcmem_free(input_buf.host);
rpcmem_deinit();
```

# Terminology – Alignment

host (ptr to image data in memory)

**<u>‘x’ dimension</u>**
min = 0, extent = 8,
stride = 1
**<u>‘y’ dimension</u>**
min = 0, extent = 8,
stride = 10

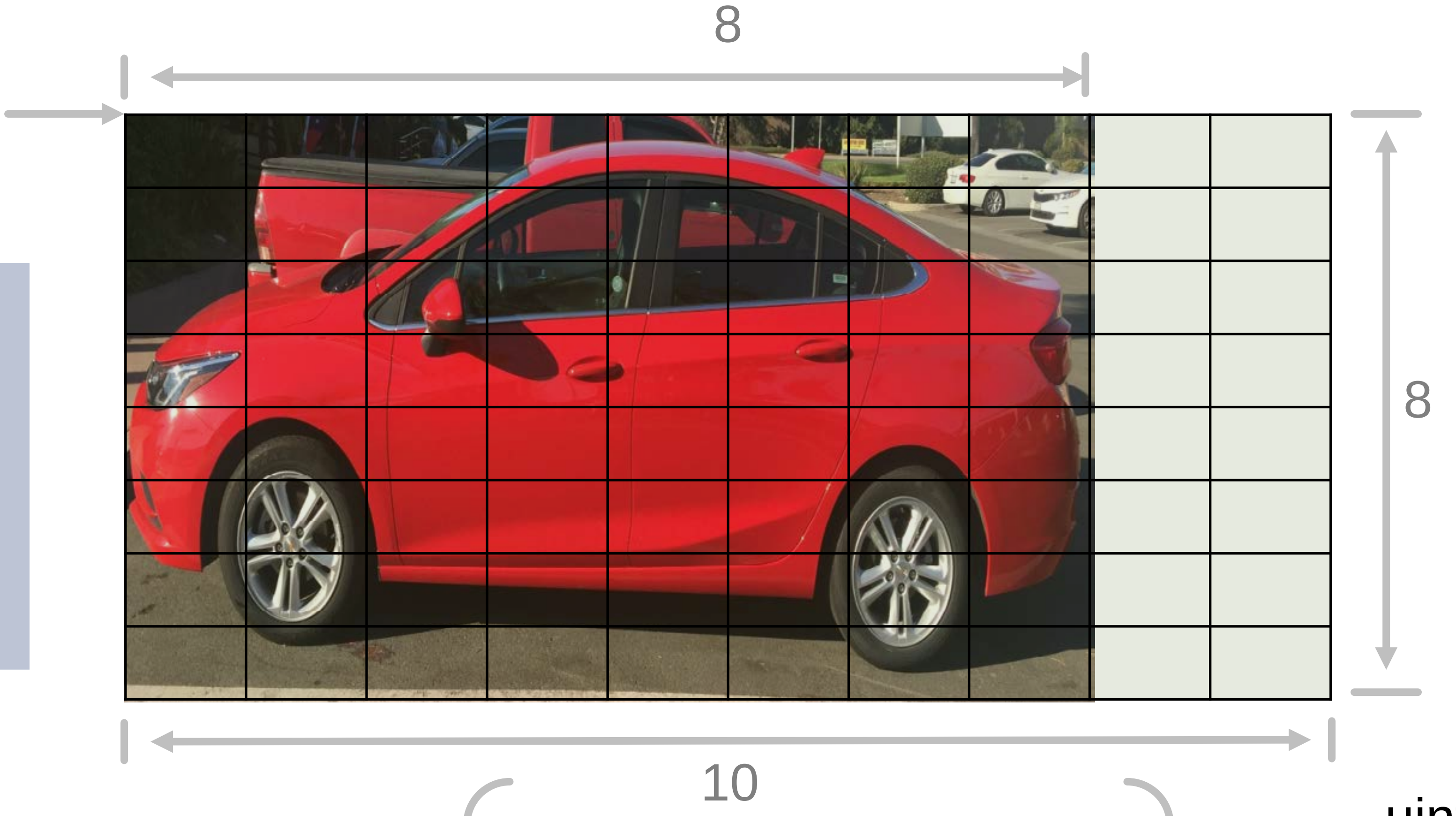


Address of pixel (x, y) = host + ( y * stride.y + x * stride.x ) * size_of(element_type)

# Terminology – Alignment (cont.)

host (ptr to the start of the buffer in memory)

**‘x’ dimension**
min = 0, extent = 8,
stride = 1
**‘y’ dimension**
min = 0, extent = 8,
stride = 10



uint8_t

Address of pixel (x, y) = host + ( y * stride.y + x *1 ) * 1

# Performance Considerations – Alignment

- On HVX, vector loads/stores from memory that are not aligned to vector width incur a performance penalty
- There are a few ways of making sure you are not paying the cost of bad alignment
  - Make sure the pipeline is compiled to generate aligned loads and stores. This is done by using two constructs
    - set_host_alignment – Use on external buffers (inputs and outputs of pipelines)

```
input.set_host_alignment(128);
output.set_host_alignment(128);
```

This is a way for the programmer to tell the Halide compiler that they guarantee that the memory is aligned. Part of the pipeline source and not the application file. **This does not mean your memory will be aligned when allocated**

- align_storage – Used to align internal memory that is allocated for a Func

```
bounded_input
        .compute_at(Func(output), y)
        .align_storage(x, 128)
        .vectorizer(x, 128, TailStrategy::Roundup);
```

Tells the Halide compiler to allocate memory for “bounded_input” in such a way that every row is aligned to a 128 byte boundary

# Performance Considerations – Alignment (cont.)

- To have aligned loads and stores generated, ensure successive rows of an image/buffer are also aligned
- In Halide, you can tell the compiler about this like so:

```
int vector_size = get_target().has_feature(Target::HVX_128) ? 128: 64;
Expr input_stride = input.dim(1).stride();
input.dim(1).set_stride((input_stride/vector_size) * vector_size);
```

If an image that violates this condition is passed to such a pipeline, the pipeline will fail an assert and crash at runtime. An example of such an image could be an image with width that is not a multiple of the vector size

This tells the Halide compiler that the programmer guarantees that the stride of the ‘y’ dimension is a multiple of the vector_size. This is a way of saying that every new row starts at an address that is aligned to the vector width

# Performance Considerations – Alignment (cont.)

- Second step to ensure that you are not paying the cost for bad alignment
  - Ensure that external buffers are allocated such that they are aligned

```
halide_device_malloc(nullptr, &input_buf, halide_hexagon_device_interface());
```

# Performance Considerations – Alignment (cont.)

- If the image width is not a multiple of vector size, then pad every row on the right (Application file)

```
const int vlen   = get_target().has_feature(Target::HVX_128) ? 128 : 64;
const int stride = (width + vlen-1)&(-vlen);  // Align rows to vector length

halide_buffer_t input_buf = {0};
halide_dimension_t in_dim[2] = {0};
halide_dimension_t out_dim[2] = {0};

input_buf.type.code = halide_type_uint;
input_buf.type.bits = 8;             // Element size in bits
input_buf.type.lanes = 1;
input_buf.dimensions = 2;

input_buf.dim = in_dim;
input_buf.dim[0].extent = width;        // Image dimensions
input_buf.dim[1].extent = height;
input_buf.dim[0].stride = 1;         // Stride for each dimension
input_buf.dim[1].stride = stride;
input_buf.dim[0].min = 0;            // Index of upper left element
input_buf.dim[1].min = 0;
```

Align the stride and allocate `stride * height * size_of(element_type)` bytes of memory

# Performance Considerations – Line Buffering

- Memory locality is an important factor that affects the latency of memory
- Common technique to improve memory latency is to use tiling
  - For a producer (f) and consumer (g) compute g in tiles, and compute f as required by tiles of g
- The problem with tiling on HVX is
  - Reasonably sized tiles are smaller than one or two vectors owing to the large vector widths supported by HVX
  - When tiling stencils, it is difficult to have the producer Funcs satisfy native vector requirements to avoid scalarization

Solution: A technique called Line Buffering
Produce rows of the producer as required by an **entire row** of the consumer

# Performance Considerations – Line Buffering (cont.)

- Consider this producer-consumer example

```
Func f, g;
f(x, y) = cast<uint16_t>(input(x-1, y)) - cast<uint16_t>(input(x+1, y));
g(x, y) = f(x, y-1) + 2*f(x, y) + f(x, y+1);
```

- Loop structure (pseudo code)

```
for(y = min_row_consumer; y <= max_row_consumer; ++y) {

            Compute producer here using “compute_at”

 for(x = min_col_consumer; x <= max_col_consumer; ++x) {
     g(x, y) = f(x, y-1) + 2*f(x, y) + f(x, y+1);
 }

}
```

In each iteration of the ‘y’ loop, one *entire row* of the consumer is computed. So, compute the producer as needed for the entire row here

# Performance Considerations – Line Buffering (cont.)

- Consider this producer-consumer example

```
Func f, g;
f(x, y) = cast<uint16_t>(input(x-1, y)) - cast<uint16_t>(input(x+1, y));
g(x, y) = f(x, y-1) + 2*f(x, y) + f(x, y+1);
```

- Loop structure (pseudo code)

```
for(y = min_row_consumer; y <= max_row_consumer; ++y) {
 int f_[3][width];
 for(y_p = y-1; y_p <= y+1; ++y) {
  for(x_p = min_col_consumer; x_p = max_col_consumer; ++x_p) {
   f_(x_p, y_p) = cast<uint16_t>(input(x_p-1, y_p)) –
                  cast<uint16_t>(input(x_p+1, y_p));
  }
 }
for(x = min_col_consumer; x <= max_col_consumer; ++x) {
     g(x, y) = f(x, y-1) + 2*f(x, y) + f(x, y+1);
 }
}
```

**f.compute_at(g, y)**

# Performance Considerations – Line Buffering (cont.)

- Consider this producer-consumer example

```
Func f, g;
f(x, y) = cast<uint16_t>(input(x-1, y)) - cast<uint16_t>(input(x+1, y));
g(x, y) = f(x, y-1) + 2*f(x, y) + f(x, y+1);
```

- Loop structure (pseudo code)

```
for(y = min_row_consumer; y <= max_row_cons
 int f_[3][width];
 for(y_p = y-1; y_p <= y+1; ++y) {
  for(x_p = min_col_consumer; x_p = max_col_consumer; ++x_p) {
   f_(x_p, y_p) = cast<uint16_t>(input(x_p-1, y_p)) –
                  cast<uint16_t>(input(x_p+1, y_p));
  }
 }
for(x = min_col_consumer; x <= max_col_consumer; ++x) {
     g(x, y) = f(x, y-1) + 2*f(x, y) + f(x, y+1);
 }
}
```

**For every row of the consumer, we can reuse two rows of the producer**

# Performance Considerations – Line Buffering (cont.)

- Loop structure (pseudo code)

```
int f[height][width];
for(y = min_row_cons
 ~~int f_[3][wid~~
 for(y_p =
  for(x_
   f_(x_

  }
 }
for(x = min_col_co
     g(x, y) = f(x, y-1) + 2
 }
}
```

```
f.store_root().compute_at(g, y)
```

# Performance Considerations – Line Buffering Summary

- Compute the producer as required by an entire row of the consumer; use “compute_at” for this
- Reuse recurring values of the producer. So, allocate storage for the producer one level above its computation; use “store_at” for this

```
Func f, g;
f(x, y) = cast<uint16_t>(input(x-1, y)) - cast<uint16_t>(input(x+1, y));
g(x, y) = f(x, y-1) + 2*f(x, y) + f(x, y+1);

// Produce lines of 'f' as required by lines of 'g’.
// Store them at root (at a higher loop level than where they are produced)
// so that they can be reused in subsequent iterations.
f.store_root().compute_at(g, y);
```

# Performance Considerations

- How wide to vectorize?
- For simple Funcs without widening, narrowing or shuffling, vectorize by the native vector size of the type of the Func, for example, if type is int16, then vectorize by 64 for HVX_128.
- For Funcs with mixed types, vectorize by the native vector size of the smallest type, for example, assuming input is an image of 8 bit pixels.

```
f(x, y) = cast<int16_t>(input(x-1, y)) + cast<int16_t>(input(x, y));
f.vectorize (x, 128) // HVX_128
```

- Downsampling should be vectorized such that the output is a native vector

**Avoid less-than-native vector types. Generating vectors that are multiples of vector size (e.g. 256 bytes in HVX_128) is absolutely fine.**

# Performance Considerations – Power APIs

- For increased control over power utilization and performance of an application, the power level can be specified before powering on HVX
- This can be accomplished by requesting a specific performance mode

```
#include "HalideRuntimeHexagonHost.h“
halide_hexagon_set_performance_mode (NULL, halide_hexagon_power_turbo);
halide_hexagon_set_performance_mode (NULL, halide_hexagon_power_nominal);
halide_hexagon_set_performance_mode (NULL, halide_hexagon_power_low);
halide_hexagon_set_performance_mode (NULL, halide_hexagon_power_default);
```

# Prefetching

- Generally, prefetching 2-3 loop iterations ahead is recommended.
- Prefetch is intended for L2 prefetching in outer loops, not within inner vectorized loops.
- To see what functions/buffers are being prefetched while generating a Halide object file, set:

  ```
  env HL_DEBUG_CODEGEN=1 …
  ```

  and then search the generated output for “Injecting prefetches…”

# Part 5 – Installation Instructions and Troubleshooting

1

Introduction to Halide

2

The Halide Programming Language

3

Halide for HVX

4

Performance Optimizations

5

Installation Instructions and Troubleshooting

# Some Details

- Halide on HVX requires Halide Tools, Hexagon SDK, and Hexagon Tools.
  - We recommend using the latest SDK version available, as of now 3.3.0 or higher
  - Get it here - https://developer.qualcomm.com/software/hexagon-dsp-sdk/tools
  - Select FULL NDK when installing
- Halide on HVX is supported on Linux and Windows
  - Hexagon tools 8.1.04
    - Linux: https://createpoint.qti.qualcomm.com/tools/#suite/961/11081
    - Windows 10: https://createpoint.qti.qualcomm.com/tools/#suite/1561/11042
  - HALIDE tools 2.0
    - Linux: https://createpoint.qti.qualcomm.com/tools/#suite/3022/13342
    - Windows 10: https://createpoint.qti.qualcomm.com/tools/#suite/3042/13341

# Android NDK – Linux Installation

- To acquire and install the android-ndk

```
$ wget http://dl.google.com/android/ndk/android-ndk-r14b-linux-x86_64.bin
$ chmod +x android-ndk-r14b-linux-x86_64.bin
$ ./android-ndk-r14b-linux-x86_64.bin
$ cd android-ndk-r14b
$ ./build/tools/make-standalone-toolchain.sh --arch=arm64
  –-install-dir=install/android-21/arch-arm64/
```

- The make-standalone-toolchain.sh step is compatible with the bash shell. If your default shell is csh or tcsh, invoke a bash subshell before running:

```
$ bash
$ ./build/tools/make-standalone-toolchain.sh --arch=arm64 --install-dir=install
```

# Android NDK – Windows Installation

- Download and install the android-ndk .
  To build Halide executables for Android, also create a standalone toolchain.py from the NDK using the make-standalone-toolchain.py script for arm64.

```
%ANDROID_ROOT_DIR%\build\tools\make_standalone_toolchain.py --arch arm64
  --api 21 --install-dir %ANDROID_ROOT_DIR%/install/android-21/arch-arm64
```

- And If needed, for ARM:

```
%ANDROID_ROOT_DIR%\build\tools\make_standalone_toolchain.py --arch arm --api 21
  --install-dir %ANDROID_ROOT_DIR%/install/android-21/arch-arm
```

- Then adjust locations specified in ANDROID_ARM64_TOOLCHAIN, SDK_ROOT, and HEXAGON_SDK_ROOT in the Halide/Examples/setup-env.cmd script:

```
C:\> cd HALIDE_Tools\2.0\Halide\Examples
C:\> setup-env.cmd
```

# Debugging with Prints – Halide print() and print_when()

- Halide provides 2 directives for printing expression values
  - print()
  - print_when()
- Can be useful when debugging an expression
- Print output is available in logcat
- Example:
  - `f(x, y) = print(x+y)`
    - This prints the value of expression x+y for all x and y
  - `f(x, y) = print_when(x = 2 && y = 4, x+y)`
    - This prints the value of x+y when x=2 and y=4

# -debug Target Feature

- To enable runtime debug output, add –debug to the target when running your generator

```
./your_generator –o . –e o,h –f pipeline target=arm-64-android-hvx_128-debug
```

- Additional output appears in ‘adb logcat’ output

```
$ adb logcat
...
01-06 23:37:36.675 17905 17905 I halide  : halide_copy_to_device 0x7fea682130 host is dirty
01-06 23:37:36.675 17905 17905 I halide  : Hexagon: halide_hexagon_device_malloc (user_context: 0x0, buf: 0x7fea682130)
01-06 23:37:36.675 17905 17905 I halide  : Hexagon: halide_hexagon_copy_to_device (user_context: 0x0, buf: 0x7fea682130)
01-06 23:37:36.675 17905 17905 I halide  : Getting device handle for interface 0x556ca620d8 device_handle 0x7f805e2060 at
addr 0x7f805e2070
01-06 23:37:36.675 17905 17905 I halide  : Getting device handle for interface 0x556ca620d8 device_handle 0x7f805e2060 at
addr 0x7f805e2070
01-06 23:37:36.675 17905 17905 I halide  :     Time: 2.375000e-02 ms
01-06 23:37:36.675 17905 17905 I halide  : Hexagon: halide_hexagon_initialize_kernels (user_context: 0x0, state_ptr:
0x556ca63040, *state_ptr: 0x0, code: 0x556ca22720, code_size: 32140)
01-06 23:37:36.675 17905 17905 I halide  :     allocating module state ->
01-06 23:37:36.675 17905 17905 I halide  :         0x7f805d10c0
01-06 23:37:36.675 17905 17905 I halide  :     halide_remote_initialize_kernels ->
01-06 23:37:36.737 17905 17905 I halide  :         79691792
01-06 23:37:36.738 17905 17905 I halide  :     Time: 6.232047e+01 ms
01-06 23:37:36.738 17905 17905 I halide  : halide_hexagon_power_hvx_on
01-06 23:37:36.738 17905 17905 I halide  :     remote_power_hvx_on ->
01-06 23:37:36.742 17905 17905 I halide  :         0
01-06 23:37:36.742 17905 17905 I halide  :     Time: 4.115990e+00 ms
```

- May slow down execution; use during development

# Initial Device Setup for Halide

- Push libraries to the device.
- If using our prebuilt binaries from the Halide release:

```
$ cd Halide/lib
$ adb push arm-32-android/libhalide_hexagon_host.so /system/lib/
$ adb push arm-64-android/libhalide_hexagon_host.so /system/lib64/
$ adb push v60/libhalide_hexagon_remote_skel.so /system/lib/rfsa/adsp/
```

- Run the executable on the device.
- If /data/local does not exist, then first time:

```
$ adb shell mkdir /data/local
$ adb push process-arm-64-android /data/local/process-arm-64-android
$ adb shell chmod 755  /data/local/process-arm-64-android
$ adb shell  /data/local/process-arm-64-android
```

# Linux Logcat

- Use logcat and mini-dm to see error messages from failed executions
- Linux:

```
$ adb shell logcat
```

- Windows:

```
C:\> adb logcat
```

# Mini-dm

- If you connect your MTP to Linux:

```
$ sudo Hexagon_SDK/3.3.0/tools/mini-dm/Linux_Debug/mini-dm
```

- If you connect your MTP to Windows:
  - Open the device manager (right click on Computer, select Device Manager).
  - Expand Ports (COM and LPT).
  - Look for "Qualcomm HS-USB Diagnostics 9025 (<port>)” and note what the <port> is (e.g. COM13).
  - Open a command prompt and start mini-dm.exe with that port

```
cd C:\Qualcomm\Hexagon_SDK\3.3.0\tools\debug\mini-dm\WinNT_Debug\
mini-dm.exe --comport COM13
```

# Performance Analysis Using Simulator Timing/Tracing

- proftool.py can be found in the SDK inside Hexagon Tools SDK/3.3.0/tools/HEXAGON_Tools/8.1.04/Tools/bin/hexagon-profiler
- Performance analysis when running with simulator:
  - Run simulator with –timing and --packet_analyze:

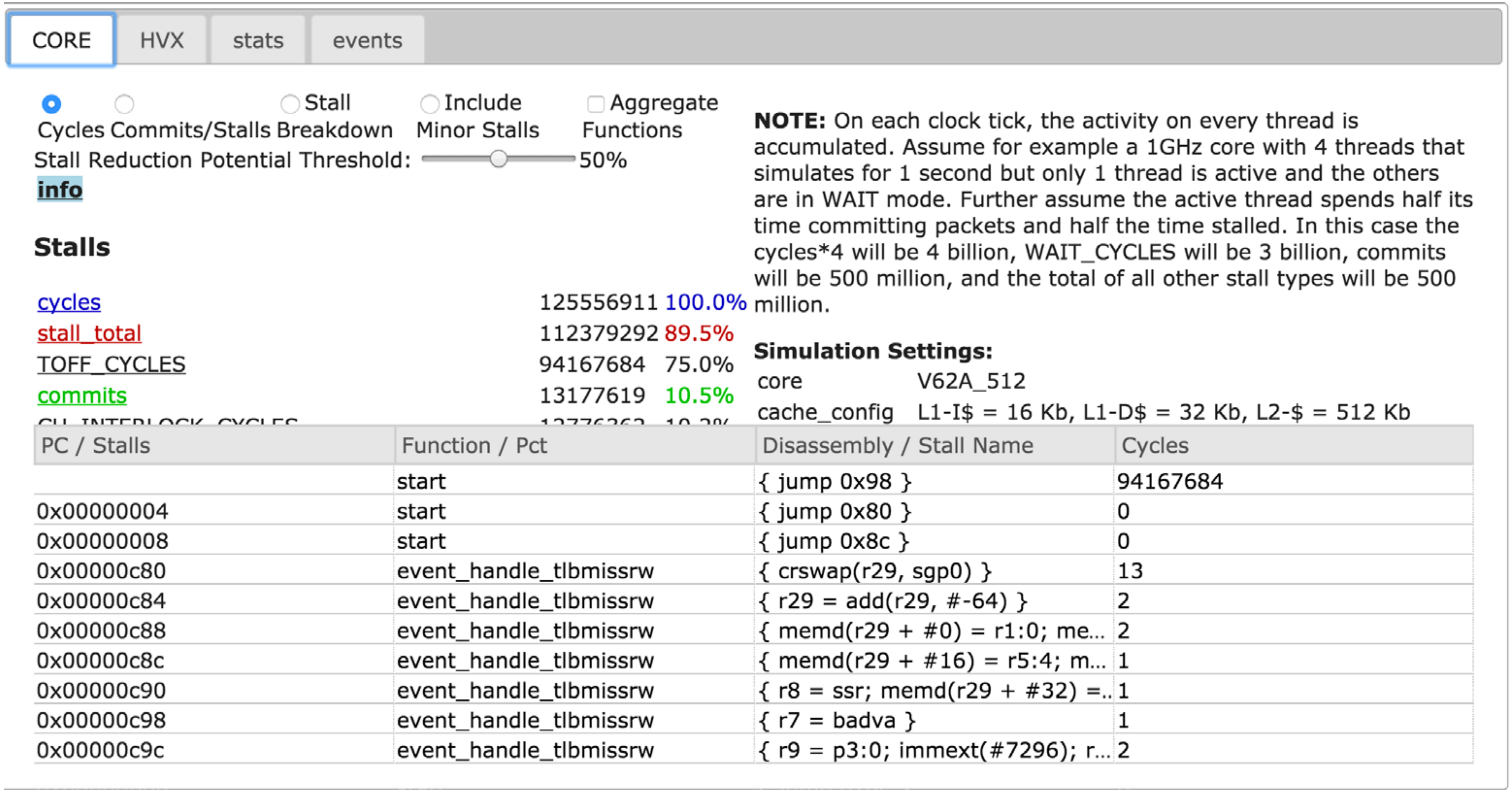
```
env TIMING=”--timing --packet_analyze process.json” hexagon-sim process
```

  - Run hexagon-profiler on the resulting stats and executable:

```
hexagon-profiler --packet_analyze --json=process.json --elf=process -o process.html
```

  - Open the generated .html in a browser to view the report

# Proftool Output (HTML)



● Cycles ○ Commits/Stalls ○ Stall Breakdown ○ Include Minor Stalls □ Aggregate Functions

Stall Reduction Potential Threshold: 50%

info

**Stalls**

| | | |
|---|---|---|
| cycles | 125556911 | 100.0% |
| stall_total | 112379292 | 89.5% |
| TOFF_CYCLES | 94167684 | 75.0% |
| commits | 13177619 | 10.5% |
| CU_INTERLOCK_CYCLES | 12776362 | 10.2% |

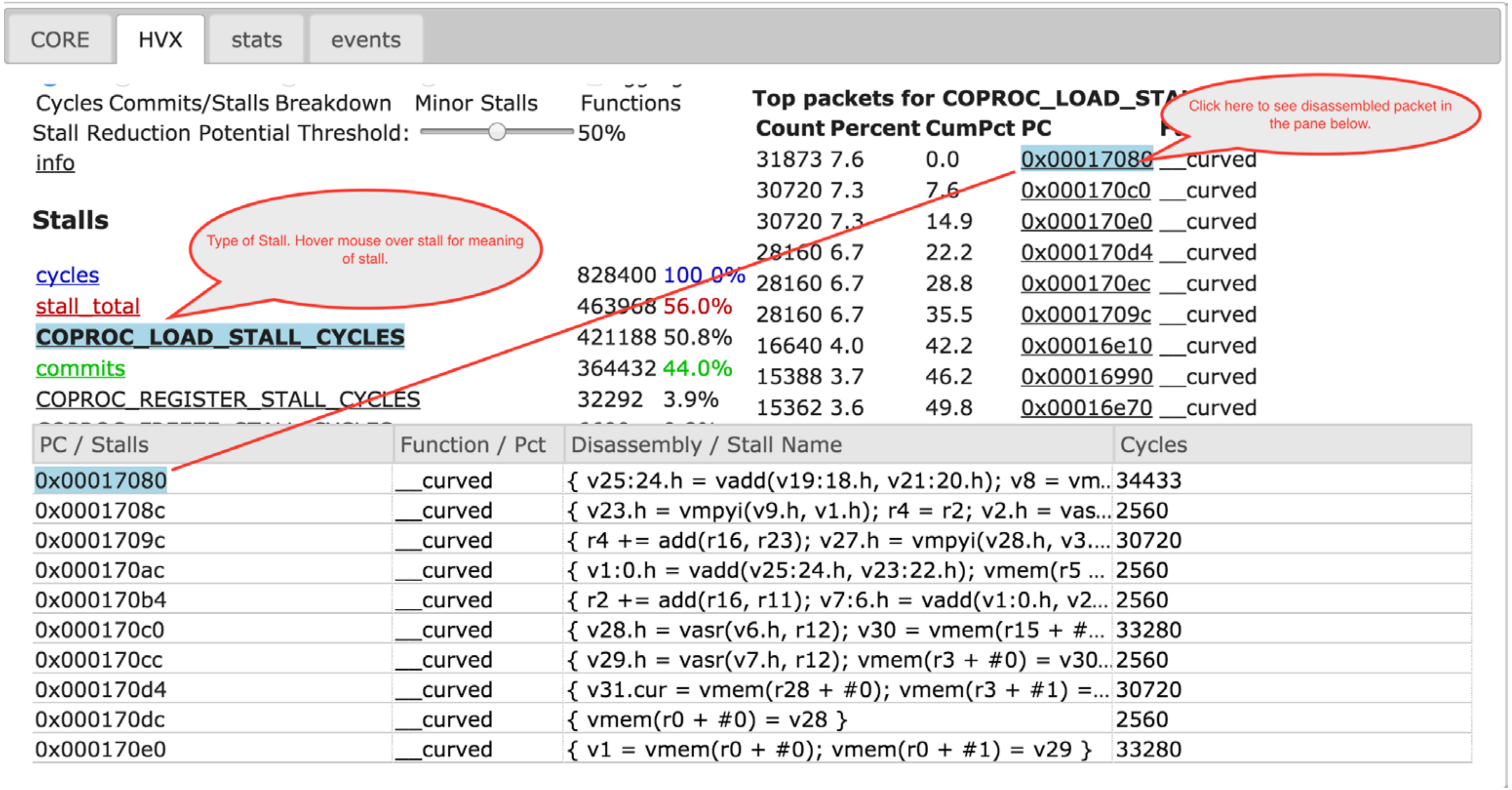
**NOTE:** On each clock tick, the activity on every thread is accumulated. Assume for example a 1GHz core with 4 threads that simulates for 1 second but only 1 thread is active and the others are in WAIT mode. Further assume the active thread spends half its time committing packets and half the time stalled. In this case the cycles*4 will be 4 billion, WAIT_CYCLES will be 3 billion, commits will be 500 million, and the total of all other stall types will be 500 million.

**Simulation Settings:**

| | |
|---|---|
| core | V62A_512 |
| cache_config | L1-I$ = 16 Kb, L1-D$ = 32 Kb, L2-$ = 512 Kb |

| PC / Stalls | Function / Pct | Disassembly / Stall Name | Cycles |
|---|---|---|---|
| | start | { jump 0x98 } | 94167684 |
| 0x00000004 | start | { jump 0x80 } | 0 |
| 0x00000008 | start | { jump 0x8c } | 0 |
| 0x00000c80 | event_handle_tlbmissrw | { crswap(r29, sgp0) } | 13 |
| 0x00000c84 | event_handle_tlbmissrw | { r29 = add(r29, #-64) } | 2 |
| 0x00000c88 | event_handle_tlbmissrw | { memd(r29 + #0) = r1:0; me... | 2 |
| 0x00000c8c | event_handle_tlbmissrw | { memd(r29 + #16) = r5:4; m... | 1 |
| 0x00000c90 | event_handle_tlbmissrw | { r8 = ssr; memd(r29 + #32) =.. | 1 |
| 0x00000c98 | event_handle_tlbmissrw | { r7 = badva } | 1 |
| 0x00000c9c | event_handle_tlbmissrw | { r9 = p3:0; immext(#7296); r... | 2 |

# Proftool Output (HTML) (cont.)



Cycles Commits/Stalls Breakdown Minor Stalls Functions

Stall Reduction Potential Threshold: 50%

info

**Stalls**

Type of Stall. Hover mouse over stall for meaning of stall.

| | | |
|---|---|---|
| cycles | 828400 | 100.0% |
| stall_total | 463968 | 56.0% |
| **COPROC_LOAD_STALL_CYCLES** | 421188 | 50.8% |
| commits | 364432 | 44.0% |
| COPROC_REGISTER_STALL_CYCLES | 32292 | 3.9% |

**Top packets for COPROC_LOAD_STA**

Click here to see disassembled packet in the pane below.

| Count | Percent | CumPct | PC | F |
|---|---|---|---|---|
| 31873 | 7.6 | 0.0 | 0x00017080 | __curved |
| 30720 | 7.3 | 7.6 | 0x000170c0 | __curved |
| 30720 | 7.3 | 14.9 | 0x000170e0 | __curved |
| 28160 | 6.7 | 22.2 | 0x000170d4 | __curved |
| 28160 | 6.7 | 28.8 | 0x000170ec | __curved |
| 28160 | 6.7 | 35.5 | 0x0001709c | __curved |
| 16640 | 4.0 | 42.2 | 0x00016e10 | __curved |
| 15388 | 3.7 | 46.2 | 0x00016990 | __curved |
| 15362 | 3.6 | 49.8 | 0x00016e70 | __curved |

| PC / Stalls | Function / Pct | Disassembly / Stall Name | Cycles |
|---|---|---|---|
| 0x00017080 | __curved | { v25:24.h = vadd(v19:18.h, v21:20.h); v8 = vm.. | 34433 |
| 0x0001708c | __curved | { v23.h = vmpyi(v9.h, v1.h); r4 = r2; v2.h = vas... | 2560 |
| 0x0001709c | __curved | { r4 += add(r16, r23); v27.h = vmpyi(v28.h, v3.... | 30720 |
| 0x000170ac | __curved | { v1:0.h = vadd(v25:24.h, v23:22.h); vmem(r5 ... | 2560 |
| 0x000170b4 | __curved | { r2 += add(r16, r11); v7:6.h = vadd(v1:0.h, v2... | 2560 |
| 0x000170c0 | __curved | { v28.h = vasr(v6.h, r12); v30 = vmem(r15 + #... | 33280 |
| 0x000170cc | __curved | { v29.h = vasr(v7.h, r12); vmem(r3 + #0) = v30... | 2560 |
| 0x000170d4 | __curved | { v31.cur = vmem(r28 + #0); vmem(r3 + #1) =... | 30720 |
| 0x000170dc | __curved | { vmem(r0 + #0) = v28 } | 2560 |
| 0x000170e0 | __curved | { v1 = vmem(r0 + #0); vmem(r0 + #1) = v29 } | 33280 |

# FAQs (cont.)

- Q: What if I don’t use .hexagon() on a pipeline stage schedule?
- A: No offloading will occur even if your target is: arm-64-android-hvx_128. Halide gives control to the programmer to offload partial pipelines

# FAQs (cont.)

- Q: I am using .hexagon() but cannot see HVX instructions in the generated assembly OR
  - How can I inspect the generated Hexagon assembly in Offload mode?
- A: You need to use .hexagon() only if you're using Offload mode. Since in Offload mode the generated assembly is either x86/arm binary depending upon whether you are using simulator/device offload. In order to inspect the halide generated hexagon assembly try using standalone mode
  - Note: you need not change your application file. Change target and remove .hexagon() and recompile

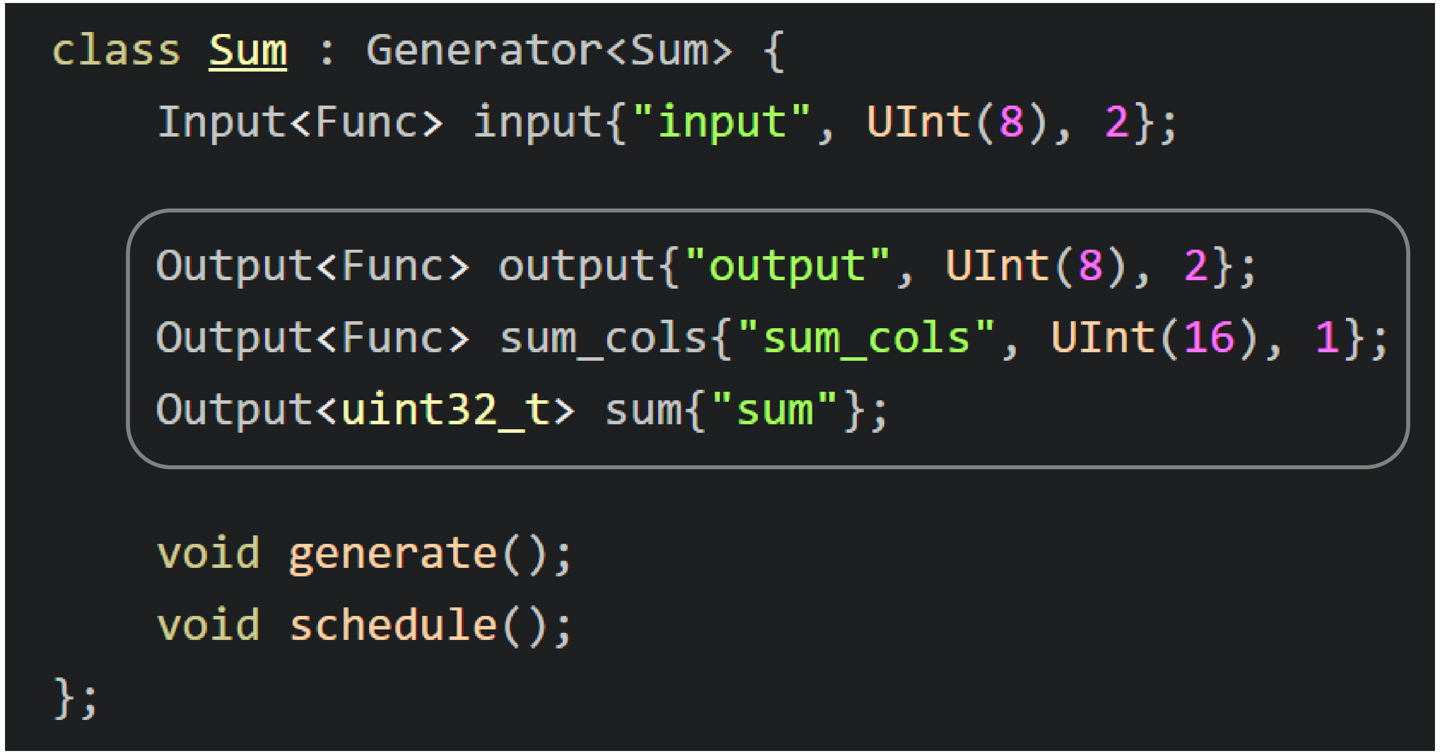
# FAQs (cont.)

- Q: Do I need to use .hexagon() and .vectorize() on every stage?
- A: No. All stages which are computed within a stage marked with .hexagon() are offloaded to hexagon. Similarly if a stage's dimension is marked with .vectorize() all other stages "inlined" within the stage are vectorized implicitly.

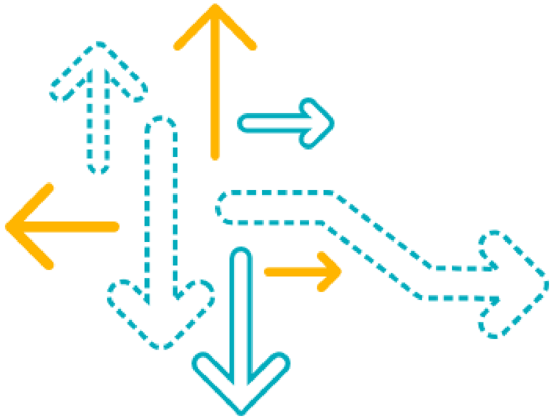
- Q: My pipeline is taking too long to compile and run. What could be the issue?
- A: This is an indicator of scalarization of code. Few rules of thumb:
  - Hexagon supports floating type arithmetic but HVX does **not**. Try converting all floating point calculations to appropriate quantized fixed point values.

# FAQs (cont.)

- Q: How do I generate a pipeline with multiple output buffers with possibly different dimensionality and sizes?
- A: Generators allow multiple outputs in the pipeline.



```
class Sum : Generator<Sum> {
    Input<Func> input{"input", UInt(8), 2};

    Output<Func> output{"output", UInt(8), 2};
    Output<Func> sum_cols{"sum_cols", UInt(16), 1};
    Output<uint32_t> sum{"sum"};

    void generate();
    void schedule();
};
```



# Questions?

https://createpoint.qti.qualcomm.com

# Acknowledgements

- We have many customers/partners engaging with Halide
  - Customer Engineering – China, India, and San Diego
  - Computer Vision – BDC, Hyderabad, and San Diego
  - GPU – BDC and San Diego
  - IOT – Toronto and San Diego
  - HQV – Toronto

# References

# References

| Documents |
|---|
| **Resources** |
| www.halide-lang.org |

| Acronyms | |
|---|---|
| **Acronym or term** | **Definition** |
| AOT | ahead-of-time |
| DSL | domain specific language |
| ILP | instruction level parallelism |
| RAW | read-after-write |