# Halide to Hardware:
## Exploring accelerators using software languages

ARM Research Summit

Jeff Setter

# Overview of the Talk

- Halide: Describes image processing and DNNs
  - CPU and GPU backends
- Hardware backend: to HLS and CGRA and SoC
- Memory Extraction: generate hardware-specialized memories
  - Including line buffers and double buffers; now unified buffers

# Halide: a data-parallel DSL

- **Algorithm**: description of computation
- **Schedule**: how to perform algorithm (efficiently)
  - Loop scheduling: `tile, split, reorder, unroll, vectorize`
  - Memory definition: `compute_at, store_at`

- Makes it easier to explore scheduling for fast execution

# Example C++ Code: 3x3 blur

3x3 Blur Kernel

| | | |
|---|---|---|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

=

| |
|---|
| 1/3 |
| 1/3 |
| 1/3 |

*

| | | |
|---|---|---|
| 1/3 | 1/3 | 1/3 |

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate blurx array

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

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

Adapted from Halide presentation by Jonathan Ragan-Kelley
Code and documentation can be found at http://halide-lang.org

# Hand-optimization adds tiling, fusing, multithreading, and vectorization to code.

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate blurx array

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

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

## Hand-optimized C++

9.9 → 0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
         a = _mm_loadu_si128((__m128i*)(inPtr-1));
         b = _mm_loadu_si128((__m128i*)(inPtr+1));
         c = _mm_load_si128((__m128i*)(inPtr));
         sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
         avg = _mm_mulhi_epi16(sum, one_third);
         _mm_store_si128(blurxPtr++, avg);
         inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

# Halide decouples the algorithm from the schedule.

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate blurx array

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

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

**3x3 blur in Halide**

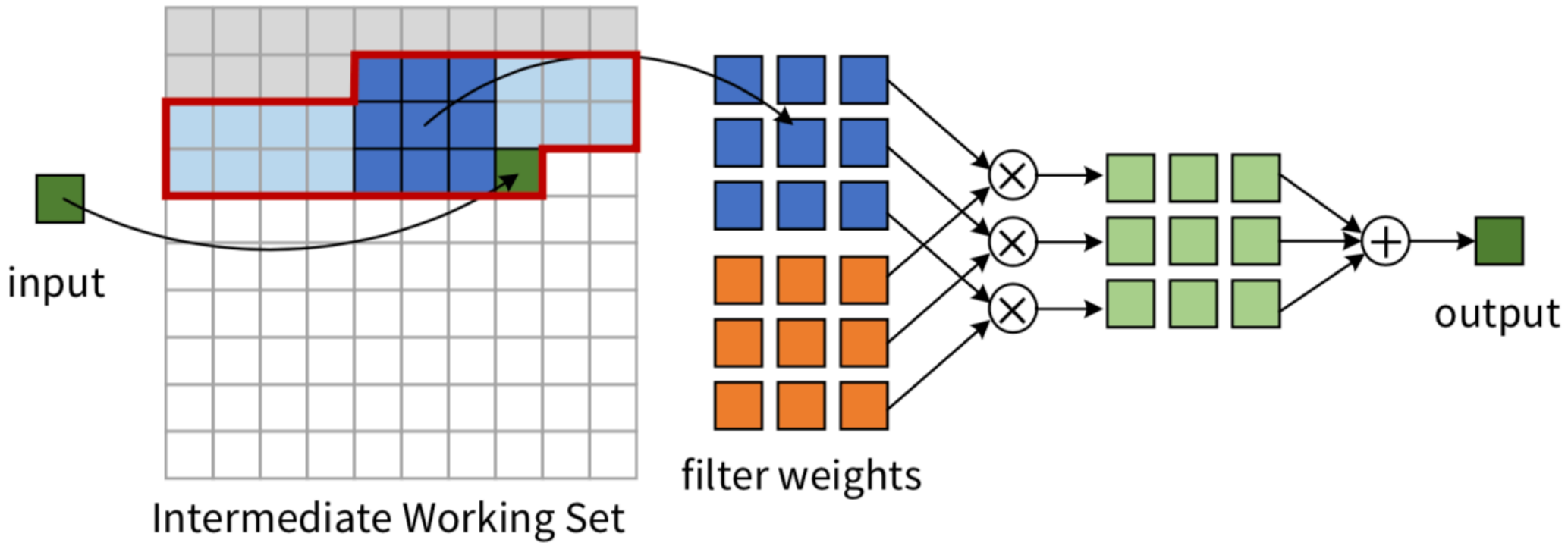
```
Var x, y, xi, yi; Func blurx, blury;
// algorithm
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

// schedule
blury.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
blurx.compute_at(blury, x).vectorize(x, 8);
```

**Hand-optimized C++**

9.9 → 0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)(inPtr-1));
          b = _mm_loadu_si128((__m128i*)(inPtr+1));
          c = _mm_load_si128((__m128i*)(inPtr));
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
          inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

# Hardware accelerator applications stream inputs in, perform computations, and stream out pixels.



# Halide Example: 3x3 convolution in hardware

```
// Algorithm
RDom win(0, 3, 0, 3);
conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);

// Schedule
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
input.in()
     .stream_to_accelerator();
     .store_at(output, xo)
     .compute_at(output, xi)

conv.update()
    .unroll(win.x)
    .unroll(win.y);
```

Defines a 3x3 convolution using input and weights.

# Halide Example: 3x3 convolution in hardware

```
// Algorithm
RDom win(0, 3, 0, 3);
conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);

// Schedule
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
input.in()
     .stream_to_accelerator();
     .store_at(output, xo)
     .compute_at(output, xi)

conv.update()
    .unroll(win.x)
    .unroll(win.y);
```

Defines a 3x3 convolution using input and weights.

Creates an accelerator from input to output operating on 64x64 image tiles.

# Halide Example: 3x3 convolution in hardware

```
// Algorithm
RDom win(0, 3, 0, 3);
conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);

// Schedule
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
input.in()
     .stream_to_accelerator();
     .store_at(output, xo)
     .compute_at(output, xi)

conv.update()
    .unroll(win.x)
    .unroll(win.y);
```

Defines a 3x3 convolution using input and weights.

Creates an accelerator from input to output operating on 64x64 image tiles.

Specifies that input should be stored in a memory.

# Halide Example: 3x3 convolution in hardware

```
// Algorithm
RDom win(0, 3, 0, 3);
conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);

// Schedule
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
input.in()
     .stream_to_accelerator();
     .store_at(output, xo)
     .compute_at(output, xi)

conv.update()
    .unroll(win.x)
    .unroll(win.y);
```

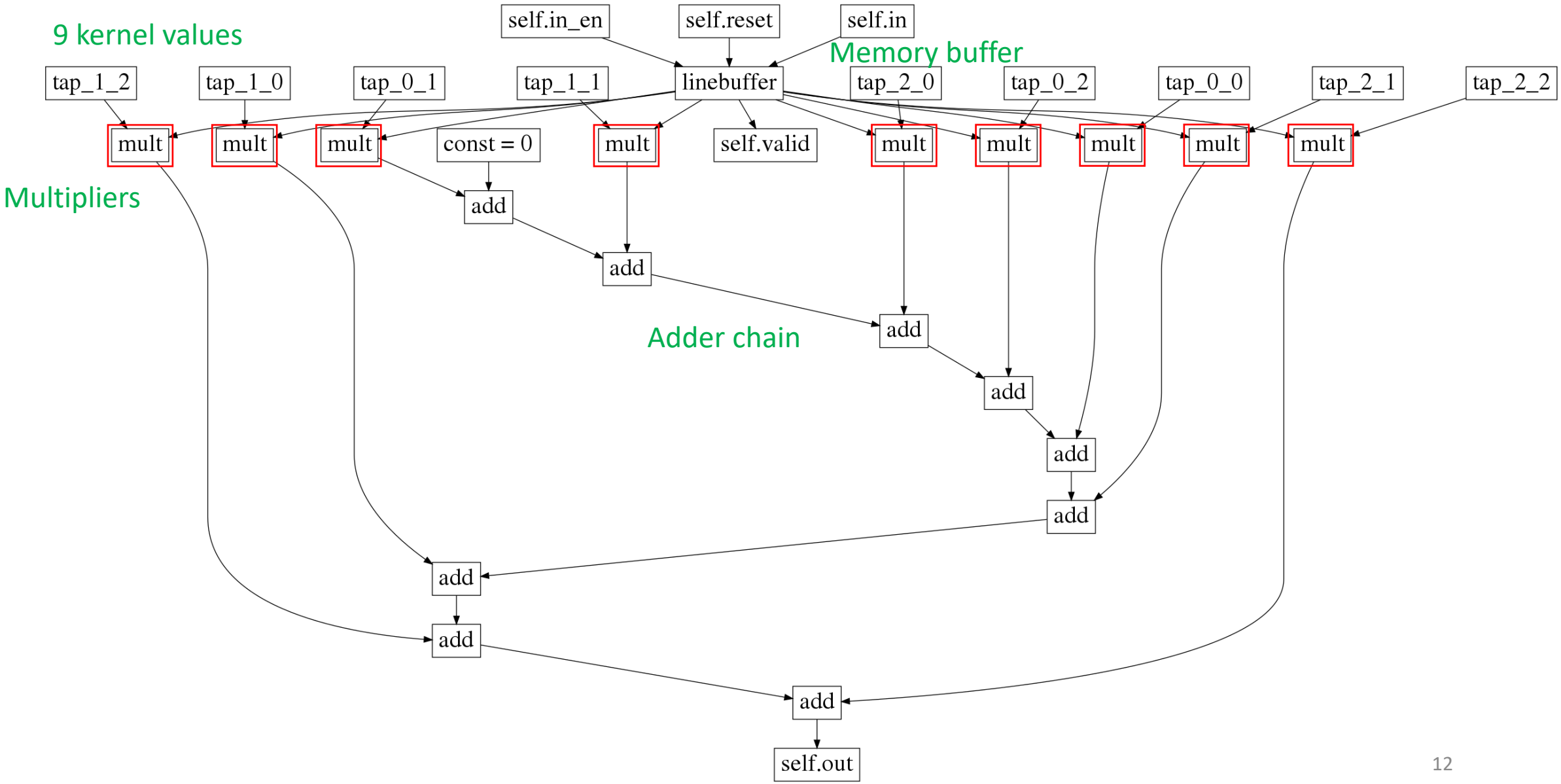
Defines a 3x3 convolution using input and weights.

Creates an accelerator from input to output operating on 64x64 image tiles.

Specifies that input should be stored in a memory.

Unrolls the implicit RDom for loops; thus 9 multipliers are created.

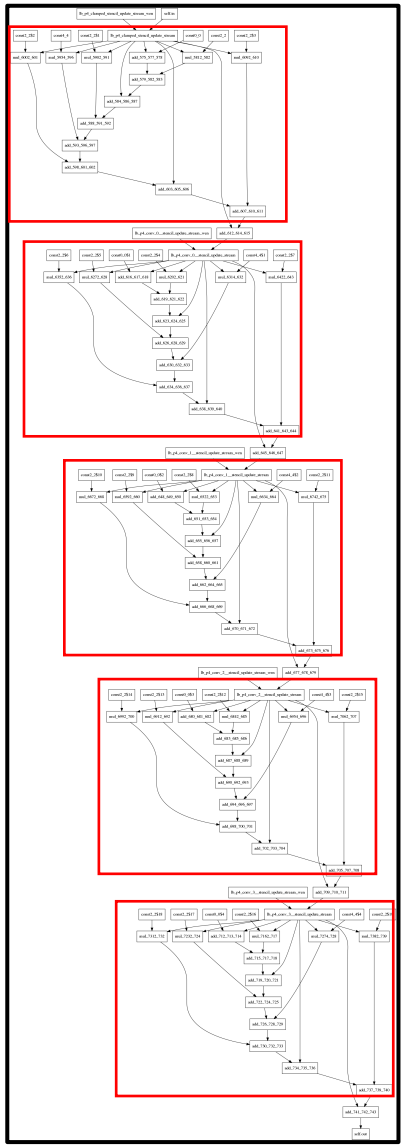
# Halide Example: 3x3 convolution hardware



| | Halide representation | Hardware IR (CoreIR) instances |
|---|---|---|
| **Input / Output** | InputParam, Param | def.input, const (set of configuration) |
| | const | const |
| **Algorithm functions** | * / + - % | mul, ashr, add, sub, and |
| | != == < <= > >= | neq, eq, {u,s}lt, {u,s}le, {u,s}gt, {u,s}ge |
| | && \|\| ! | and, or, not |
| | & \| ~ ^ >> << | and, or, not, xor, {a,l}shr, shl |
| | select max min | mux, {u,s}max, {u,s}min |
| | absd, * + | absd, mad |
| **Floating Point** | *[float]* * / + - % | fmul, fdiv, fadd, fsub, frem |
| | *[float]* != == < <= > >= | fneq, feq, flt, fle, fgt, fge |
| | select min max floor ceil | fmux, fmin, fmax, fflr, fceil |
| | log exp pow sqrt | log, exp, pow, sqr |
| | sin cos tan asin acos atan2 | sin, cos, tan, asin, acos, atan |
| **Control flow** | for, if | counter, *enable wire* |
| | *var load linebuffer stencil*,<br>*var load array* | *input* => muxn,<br>*const* => muxn |
| **Schedule primitives** | accelerate | Create circuit between input and output |
| | linebuffer = comp+store_at | Create linebuffer (memories and registers) |
| | RDom | Define stencil input size for linebuffer |
| | unroll | Duplicate algorithm operators by amount.<br>Can be used to remove counters / var load. |
| | tile | Define linebuffer width |
| | reorder | Define how data is read from image |

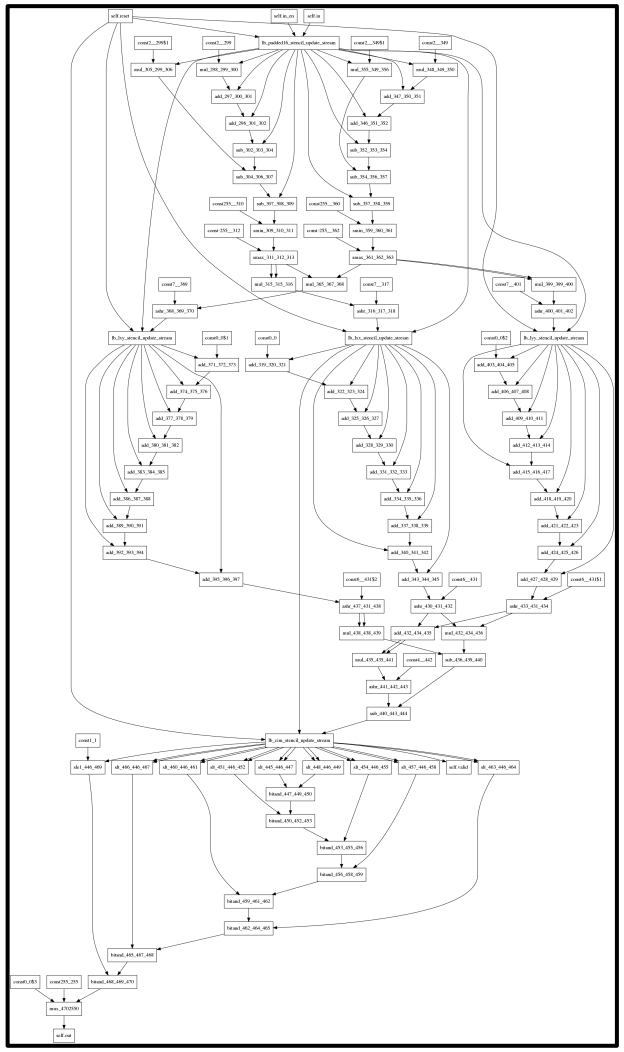
# conv_chain: successive convolutions

```
#define NUM_CONV 5
#define WIN_SIZE 3

Var x, y, xi, yi, xo, yo; Func conv[NUM_CONV]; Func out;
RDom win(0,3, 0,3);
kernel(x,y) = {{1,2,1},{2,4,2},{1,2,1}};

// algorithm
for (uint i = 0; i < NUM_CONV; ++i) {
    if (i == 0) {
      conv[i](x, y) += input(x+win.x, y+win.y) *
                  kernel(win.x, win.y);
    } else {
      conv[i](x, y) += conv[i-1](x+win.x, y+win.y) *
                          kernel(win.x, win.y);
    }
}
out(x, y) = conv[NUM_CONV-1](x, y);

// schedule
for (i=0:NUM_CONV) { conv[i].update(0).unroll(win.x).unroll(win.y);
conv[i].linebuffer(); }
out.tile(x,y, xo,yo, xi,yi, 62,62).reorder(xi,yi, xo,yo);
```



# harris: corner detection algorithm

```
// algorithm (without shifts for 16 bits)
// sobel filter
padded16(x,y) = padded(x,y);
grad_x(x, y) =  - padded16(x-1,y-1) +  padded16(x+1,y-1)
               -2*padded16(x-1,y)   + 2*padded16(x+1,y)
                - padded16(x-1,y+1) +  padded16(x+1,y+1));
grad_y(x, y) = padded16(x-1,y+1) – padded16(x-1,y-1)
             + 2*padded16(x,y+1) - 2*padded16(x,y-1)
             + padded16(x+1,y+1) - padded16(x+1,y-1));
// product of gradients
grad_xx(x, y) = grad_x(x,y) * grad_x(x,y);
grad_yy(x, y) = grad_y(x,y) * grad_y(x,y);
grad_xy(x, y) = grad_x(x,y) * grad_y(x,y));
// box filter (i.e. windowed sum)
grad_gx(x, y)  += grad_xx(x+box.x, y+box.y);
grad_gy(x, y)  += grad_yy(x+box.x, y+box.y);
grad_gxy(x, y) += grad_xy(x+box.x, y+box.y);
// calculate corner measure (Cim)
Expr lgx  = grad_gx(x, y)  >> 7;
Expr lgy  = grad_gy(x, y)  >> 7;
Expr lgxy = grad_gxy(x, y) >> 7;
Expr det = lgx*lgy - lgxy*lgxy;
Expr trace = lgx + lgy;
cim(x, y) = det - (trace*trace >> 4);
// Perform non-maximal suppression (nms)
Expr is_max = cim(x,y) > cim(x-1,y-1) && cim(x,y) > cim(x,y-1)
           && cim(x,y) > cim(x+1,y-1) && cim(x,y) > cim(x-1,y)
           && cim(x,y) > cim(x+1,y)   && cim(x,y) > cim(x-1,y+1)
           && cim(x,y) > cim(x,y+1)   && cim(x,y) > cim(x+1,y+1);
hw_output(x, y) = select( is_max && (cim(x, y) >= threshold),
                          255, 0);

// schedule
grad_{x,y,xx,yy,xy,gx,gy,gxy}.linebuffer().unroll(x);
grad_{gx,gy,gxy}.update(0).unroll(x).unroll(box.x).unroll(box.y);
```

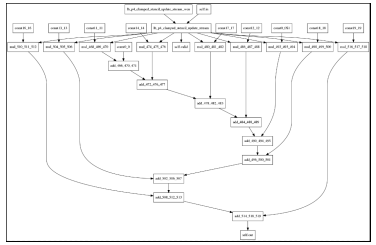


# Rough overview of CGRA application flow

```
Var x, y, xi, yi, xo, yo; Func conv, out;
RDom win(0,3, 0,3);
kernel(x,y) = {{11,12,13},{14,15,16},{17,18,19}};

// algorithm
conv(x, y) += input(x+win.x, y+win.y) *
                kernel(win.x, win.y);
out(x, y) = conv(x, y);

// schedule
conv.update(0).unroll(win.x).unroll(win.y);
out.tile(x,y, xo,yo, xi,yi, 62,62).reorder(xi,yi, xo,yo);
conv.linebuffer();
```

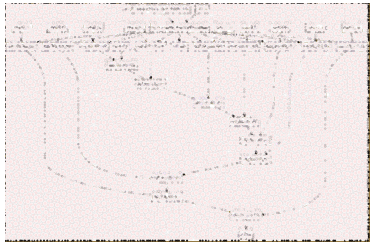
Application DSL: Halide

Compiler →



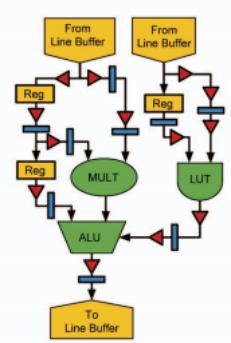
Hardware IR: CoreIR

Tech Mapper →



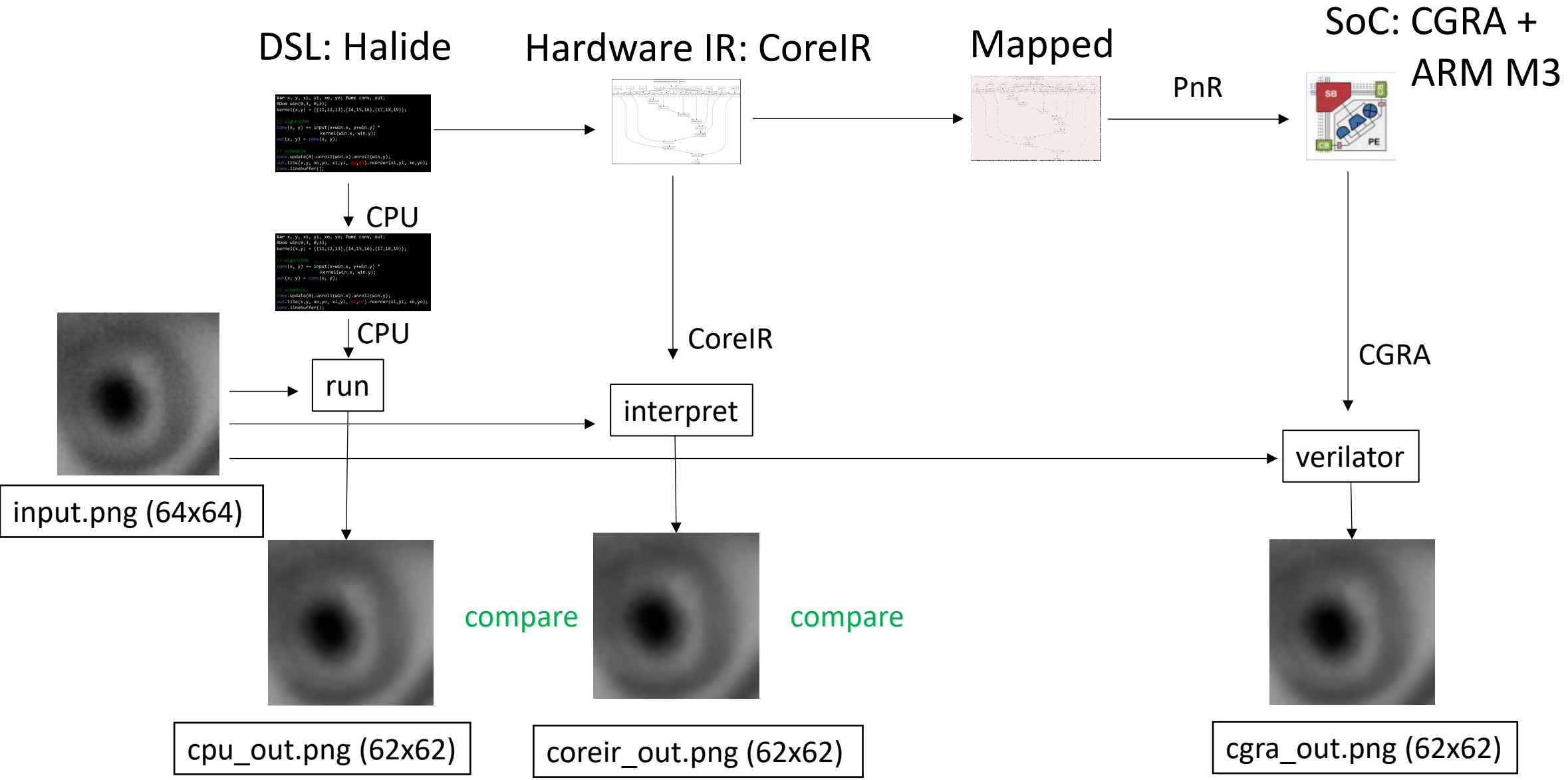
Halide IR: Mapped CoreIR

PnR →

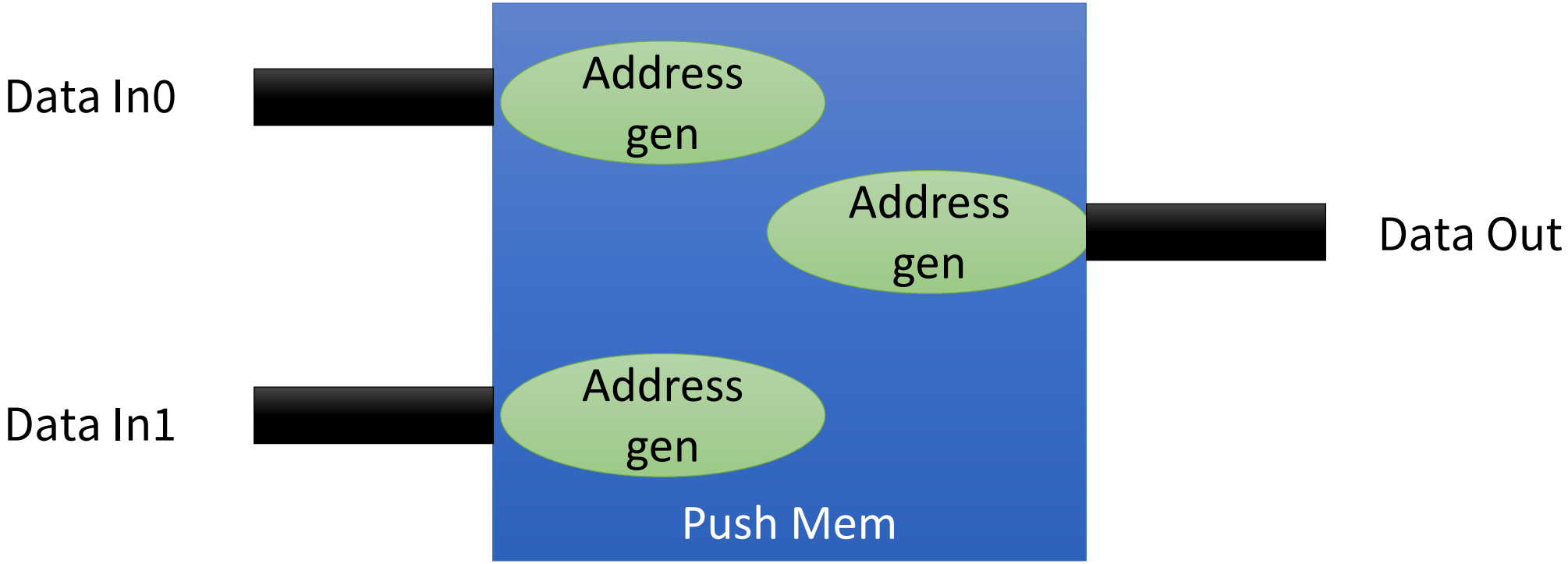



CGRA bitfile

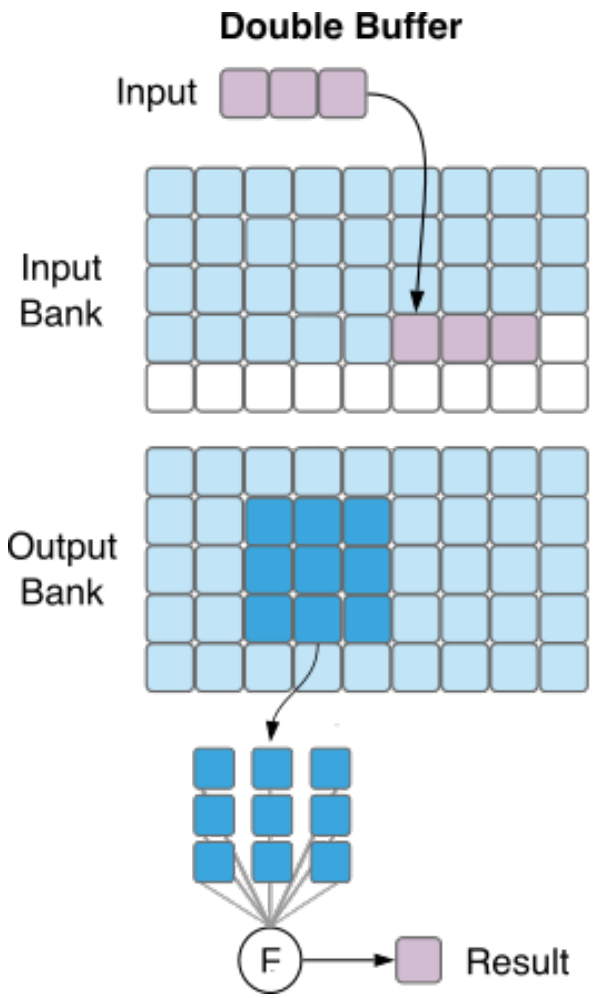
# Testing - Image Testing for Applications



Common difficulty: designing and configuring memories.
Our solution: a unified buffer.



# Example 1 - Double buffer:
# Loading next frame while computing current frame.

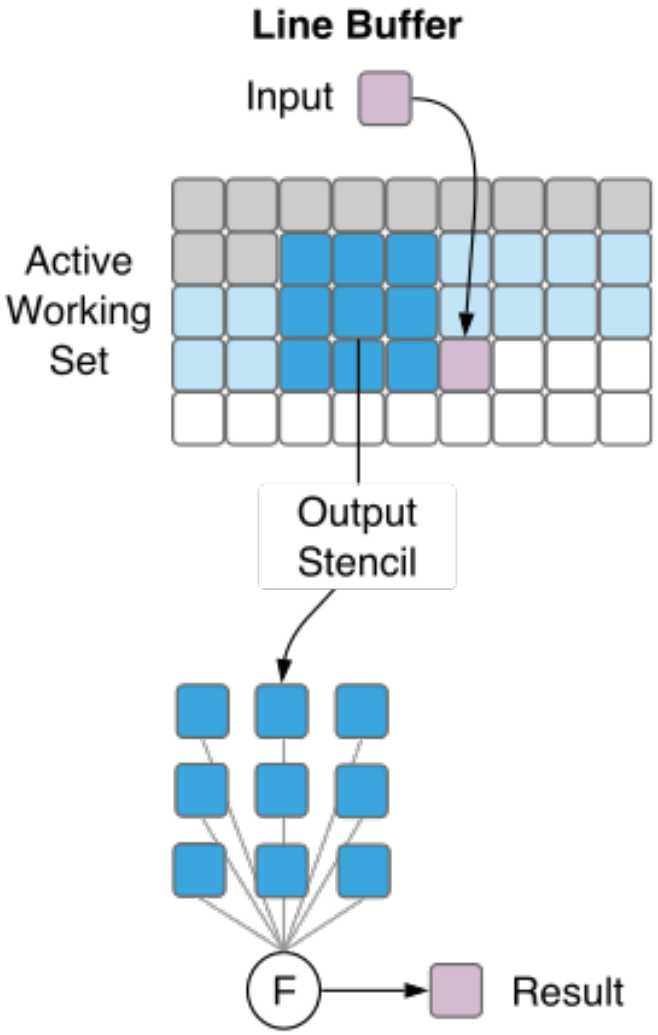



Input chunk size: [9, 5]
Input block size: [3, 1]

Output stencil size: [9, 5]
Output block size: [3, 3]

- Input chunk is the same size as the output stencil
- In this example, the output stencil is too large (45 pixels), so only an output block is read each cycle (9 pixels)
- For a double buffer, we **access** the buffer **over multiple cycles**

# Example 2 - Line buffer:
# Handling raster-order data with locality.



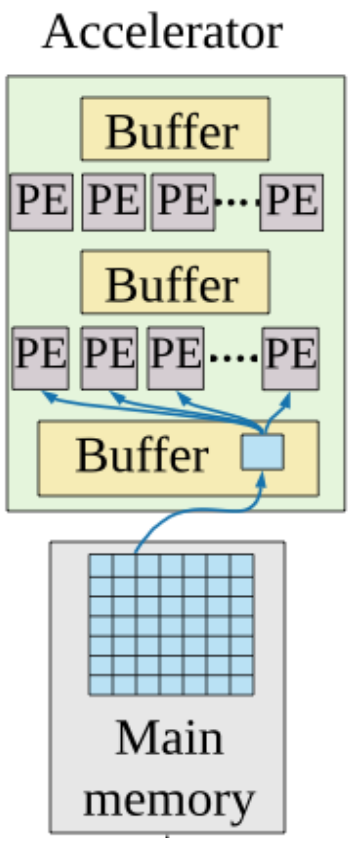

Input chunk size: [1, 1]

Logical buffer size: [9, 5]
Output stencil size: [3, 3]

- Input and output blocks are the same size as the chunk/stencil
- Since the input is just 1 pixel, then 8 pixels in the new output stencil are reused
- For a line buffer, we have **<u>overlapping output stencils</u>**

# Example: MobileNet in Halide

```
// Algorithm
RDom r_dw(-1,3, -1,3), r_pw(0, 32);
dw_conv(x, y, c) += input(x+r_dw.x, y+r_dw.y, c)
                    * w1(1 + r_dw.x, 1 + r_dw.y, c);
pw_conv(x, y, k) += dw_conv(x,y, r_pw.c)
                   * w2(r_pw.c, k);
// Schedule
pw_conv.split(x, xo, xi, 16)
       .split(y, yo, yi, 16)
       .reorder(k, xi, yi, r_pw.c, xo, yo);
dw_conv.reorder(x, y, c);
dw_conv.compute_at(pw_conv, xi)
        .store_at(pw_conv, xo)
pw_conv.accelerate({input}, xo)
dw_conv.unroll(r_dw.x, 3).unroll(r_dw.y, 3);
pw_conv.unroll(k, 32);
```




Tiled

Accelerator with buffers

Parallel compute

# MobileNet UBuffers

| | Logical | Output stencil | Output block | Input chunk | Input block |
|---|---|---|---|---|---|
| input | 18 x 18 x 32 | 3 x 3 x 1 | 3 x 3 x 1 | 1 x 1 x 1 | 1 x 1 x 1 |
| dw_conv | 16 x 16 x 32 | 16 x 16 x 32 | 1 x 1 x 1 | 16 x 16 x 32 | 1 x 1 x 1 |
| pw_conv | 16 x 16 x 32 | 16 x 16 x 32 | 1 x 1 x 32 | 16 x 16 x 32 | 1 x 1 x 1 |

```
allocate output(112, 112, 32)
for pw.yo: 0 -> 7
  for pw.xo: 0 -> 7  // store level: input, dw_conv, pw_conv
                     // compute level: input, pw_conv
    allocate input(18, 18, 32)
    allocate dw_conv(16, 16, 32)
    allocate pw_conv(16, 16, 32)
    for r_pw.c: 0 -> 32
      for pw.yi: 0 -> 16
        for pw.xi: 0 -> 16   // compute level: dw_conv
          unrolled for r_dw.y: 0 -> 3
            unrolled for r_dw.x: 0 -> 3
              dw_conv(pw.xi, pw.yi, r_pw.c) +=
                input(pw.xi + r_dw.x, pw.yi + r_dw.y, r_pw.c) *
                w1(r_dw.x, r_dw.y, r_pw.c)

          unrolled for pw.k: 0 -> 32
              pw_conv(pw.xi, pw.yi, pw.k) +=
                dw_conv(pw.xi, pw.yi, r_pw.c) *
                w2(r_pw.c, pw.k)
```

Three Unified Buffers

# MobileNet UBuffers

| | Logical | Output stencil | Output block | Input chunk | Input block |
|---|---|---|---|---|---|
| input | 18 x 18 x 32 | 3 x 3 x 1 | 3 x 3 x 1 | 1 x 1 x 1 | 1 x 1 x 1 |
| dw_conv | 16 x 16 x 32 | 16 x 16 x 32 | 1 x 1 x 1 | 16 x 16 x 32 | 1 x 1 x 1 |
| pw_conv | 16 x 16 x 32 | 16 x 16 x 32 | 1 x 1 x 32 | 16 x 16 x 32 | 1 x 1 x 1 |

```
allocate output(112, 112, 32)
for pw.yo: 0 -> 7
  for pw.xo: 0 -> 7  // store level: input, dw_conv, pw_conv
                     // compute level: input, pw_conv
    allocate input(18, 18, 32)
    allocate dw_conv(16, 16, 32)
    allocate pw_conv(16, 16, 32)
    for r_pw.c: 0 -> 32
      for pw.yi: 0 -> 16
        for pw.xi: 0 -> 16    // compute level: dw_conv
          unrolled for r_dw.y: 0 -> 3
            unrolled for r_dw.x: 0 -> 3
              dw_conv(pw.xi, pw.yi, r_pw.c) +=
                input(pw.xi + r_dw.x, pw.yi + r_dw.y, r_pw.c) *
                w1(r_dw.x, r_dw.y, r_pw.c)

          unrolled for pw.k: 0 -> 32
              pw_conv(pw.xi, pw.yi, pw.k) +=
                dw_conv(pw.xi, pw.yi, r_pw.c) *
                w2(r_pw.c, pw.k)
```

# Halide to Hardware

- Compiler analyses help optimize the backend, such as extracting specialized hardware memories for our CGRA.
- By separating the algorithm from the schedule, CPU and hardware implementations can share much of the same code.
- We can make it easier to get others to create hardware, by extending compiler for DSLs like Halide.