Qualcomm Technologies, Inc.

Halide for HVX

User Guide

80-PD002-1 Rev. C

November 21, 2018

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

Revision history

Revision	Date	Description
A	June 2017	Initial release
B	October 2017	Updated for Halide 2.0. Numerous changes were made to this document; it should be read in its entirety.
C	November 2018	Numerous changes were made to this document; it should be read in its entirety.

Contents

Tables

1 Introduction

This document describes Halide usage for the Qualcomm® Hexagon™ Vector eXtensions (HVX) compiler. Halide is a Domain Specific (DSL) programming language (a dialect of C++) for image processing.

Qualcomm Halide for HVX is used to target the HVX architecture:

- As a compiler for generating code to run on Hexagon with HVX
- As runtime libraries to support running code compiled by the Halide compiler on the Hexagon with HVX

Halide for Hexagon Vector eXtensions (HVX) allows developers to utilize features of the HVX processor without knowledge of the underlying HVX architecture. It also supports offloading of workload to the Qualcomm Hexagon DSP on Qualcomm® Snapdragon™ devices.

For more information and resources on the Halide programming language, see http://halide-lang.org/.

1.1 Conventions

Square brackets enclose optional items (for example, `[label]`).

The vertical bar character, `|`, indicates a choice of items (for example, `add|del`).

Function declarations, function names, type declarations, attributes, and code samples appear in a different font (for example, `cp armcc armcpp`).

Code variables appear in angle brackets (for example, *<number>*).

1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

For questions about Halide for HVX, email halide@quicinc.com.

2 Getting started with Halide for HVX

2.1 Download the Hexagon SDK

Install the following:

- Hexagon LLVM toolset
- Halide toolset
- Hexagon SDK

1. Go to https://developer.qualcomm.com/software/hexagon-dsp-sdk/tools.
2. Locate and download version 3.3.0 for Linux under the **Hexagon Series 600 Software** heading.
3. Untar the installer.
4. Run the extracted installer to install the Hexagon SDK and Tools selecting **Installation of Hexagon SDK** and designating `/location/of/SDK/Hexagon_SDK/3.3.0` as the filepath.

2.2 Install prerequisite packages

For the camera_pipe use case, Halide on Linux requires the following libraries to be installed on the host:

- sudo apt-get install libjpeg-dev
- sudo apr-get install libpng12-dev
- sudo apt-get install zlib1g-dev

2.3 Set up the environment

Set environment variables to point to the SDK installation locations:

```
export SDK_ROOT=/location/of/SDK
export HEXAGON_SDK_ROOT=$SDK_ROOT/Hexagon_SDK/<sdk_rev>
export HL_HEXAGON_TOOLS=$HEXAGON_SDK_ROOT/tools/HEXAGON_Tools/<tools_rev>/
Tools
export HALIDE_ROOT=/location/of/HALIDE_Tools/<halide_rev>/Halide
```

`HL_HEXAGON_TOOLS` should point to the Tools directory of the Hexagon LLVM toolset.

2.4 Build and run a Halide for HVX example

Build and run an example on the simulator

Halide supports running Hexagon code on the simulator from the Hexagon tools.

Build and run the dilate-3x3 example on the simulator:

```
cd $HALIDE_ROOT/Examples/dilate3x3
./test-dilate3x3
```

Build and run an example on a Snapdragon device

Halide uses a small runtime library that must be present on the device.

```
adb shell mkdir -p /system/lib/rfsa/adsp
adb push $HALIDE_ROOT/lib/arm-32-android/libhalide_hexagon_host.so /system/
lib/
adb push $HALIDE_ROOT/lib/arm-64-android/libhalide_hexagon_host.so /system/
lib64/
adb push $HALIDE_ROOT/lib/v60/libhalide_hexagon_remote_skel.so /system/lib/
rfsa/adsp/
```

The libhalide_hexagon_remote_skel.so library must be signed or the device must be signed as a debug device to run Hexagon code. See `Hexagon_SDK/3.4.1/docs/Tools_Signing.html` for more information about signing Hexagon binaries.

The `libhalide_hexagon_remote_skel.so` library must be signed or the device must be signed as a debug device to run Hexagon code.See `Hexagon_SDK/3.0/docs/Tools_Signing.html` for more information about signing Hexagon binaries.

To build a Halide example – Halide/Examples/blur – for execution on a Snapdragon device, do the following:

1. Create a standalone toolchain from the Android NDK with the `make-standalone-toolchain.sh` script.

   ```
   export ANDROID_NDK_HOME=$HEXAGON_SDK_ROOT/tools/android-ndk-r14b/
   export ANDROID_ARM64_TOOLCHAIN=$ANDROID_NDK_HOME/install/android-21/arch-
   arm64
   $ANDROID_NDK_HOME/build/tools/make--standalone--toolchain.sh --arch=arm64
   …. --platform=android-21 --install-dir=$ANDROID_ARM64_TOOLCHAIN
   ```

2. Run the example script provided.

   ```
   export RUNDROID=1
   export ANDROID_SERIAL=<device_id>
   cd $HALIDE_ROOT/Examples/blur
   ./test-Blur-app-android
   ```

To build a Halide example on Windows in the Examples directory:

```
setup-env.cmd
cd %HALIDE_ROOT%\Examples\blur test-Blur-app-android.cmd
```

3 Programming with Halide for HVX

3.1 Halide for HVX Runtime Model

Halide pipelines are compiled by the user to generate a binary, which is executed at runtime. Halide for HVX supports two runtime targets:

- Snapdragon devices
- Hexagon simulator

3.2 Halide execution modes

Programmers can generate two variants of Halide binaries:

- Offload mode – The generated Halide binary is launched from a host processor.
  - On Snapdragon devices, the host process is the ARM applications processor. This runtime model is also called the Device-offload mode.
  - On the hexagon simulator runtime target, the host processor is the x86 processor. This runtime model is called the Simulator-offload mode.
- Standalone mode – The generated Halide binary is a standalone HVX object file. This file can be used to integrate Halide programs into an existing vision pipeline.
  - On Snapdragon devices, the generated object file can be launched on the HVX processor. This runtime model is called the Device-standalone model.
  - On the Hexagon simulator runtime target, the generated object file can be used to simulate an end-to-end vision algorithm running on the Hexagon DSP with HVX. This runtime model is called the Simulator-standalone model.

3.3 Author and build your first Halide program

To start authoring code with Halide, modify an existing example for the Hexagon simulator runtime target (Simulator-standalone mode). For instance, use an editor to open the following file:

```
$HALIDE_ROOT/Examples/gaussian5x5/gaussian5x5.cpp
```

The examples are intended as templates to be modified as needed.

To build and run the Halide program on the simulator, invoke `test-gaussian5x5` on Linux or `test-gaussian5x5.cmd` on Windows. The compilation model in Halide for HVX builds a Halide executable in two steps. this is also called ahead-of-time (AOT) compilation.

1. The native, x86 compiler is invoked on the Halide sources to build an x86 executable. This is called the Generator.
2. The Generator is executed to produce a HVX binary.

   NOTE In the rest of the document, the terms Generator and Halide are used interchangeably.

The test-gaussian5x5 script runs both these steps to produce an HVX binary and then invokes the Hexagon simulator on the HVX binary.

4 Introduction to the Halide programming language

Halide is a domain-specific language for vision and image processing algorithms and allows programmers to author efficient image processing pipelines. Halide separates the program into two conceptual parts:

- The algorithm – Defines what is computed at each pixel
- The schedule – Defines how the computation should be organized

Every Halide program must consist of these two parts, and the user specifies both of them. The following example is a horizontal blur program authored in Halide:

```
// Image with 8 bits per pixel.
ImageParam input(UInt(8), 2) Halide::Func f;
// Algorithm.
f(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
// Schedule
f.hexagon().vectorize(x, 8).parallel(y, 16);
```

The algorithm concisely defines the blur computation in terms of the x and y coordinates of a pixel. The schedule directs how the computation should be conducted. In this instance, the programmer has directed Halide to vectorize the blur algorithm by a factor of 8 pixels, parallelize the algorithm by a factor of 16, and launch it on the Hexagon HVX processor.

The Halide HVX compiler automatically translates the high-level image algorithm written by the programmer into an efficient HVX executable.

The Halide community has a rich set of tutorials hosted at https://halide-lang.org/tutorials/tutorial_introduction.html. An in-depth explanation of the different features of the language is beyond the scope of this document. However, this section presents a high-level introduction to some of the important features of the language.

4.1 Algorithm

Every Halide program has two parts, an algorithm and a schedule. The algorithm defines what is computed at each pixel. Here we present the constructs that Halide provides to write an algorithm.

Func

A Func represents a pipeline stage. It describes what value a pixel should have. A Func is backed by a memory buffer, which can be thought of as an image. In the following example, `f` is a Func that is used to compute an image where every pixel is the sum of its x and y coordinates.

```
f(x, y) = x + y;
```

A Func can be an input to another Func. In this case, an image processing pipeline can be constructed by cascading multiple Funcs (each of which is a pipeline stage) in producer-consumer relationships. The following example is a Gaussian Blur filter represented in Halide:

```
bounded_input(x, y) = BoundaryConditions::repeat_edge(input)(x, y);
input_16(x, y) = cast<int16_t>(bounded_input(x, y));
input_16(x, y) = cast<int16_t>(input(x, y));
rows(x, y) = input_16(x, y-2) + 4*input_16(x, y-1) + 6*input_16(x,y)+
             4*input_16(x,y+1) + input_16(x,y+2);

cols(x,y) =  rows(x-2, y) + 4*rows(x-1, y) + 6*rows(x, y) +
             4*rows(x+1, y) + rows(x+2, y);
output(x, y)  = cast<uint8_t> (cols(x, y) >> 8);
```

`Bounded_input`, `input_16`, `rows`, `cols`, and `output` are all Funcs, each representing a stage of this 5x5 Gaussian blur filter. `Output` is the final stage of the pipeline, and the memory buffer that is associated with output contains the blurred image.

Var

A Var is a name to use as a variable when defining a Func. A Func can be defined as `Var x,y:`

```
blur_x(x , y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
```

In this example, `x` and `y` have no meaning on their own. However, when used with the `blur_x` and `input` Funcs, they signify the two dimensions of these Funcs.

Expr

An Expr in Halide is composed of Funcs, Vars, and other Exprs, for example: of Exprs in Halide.

```
Expr e = x + y;
Output(x, y) = 3*e + x;
```

In these examples, `x` and `y` are Vars, `e` is an Expr, and `Output` is a Func.

4.2 Schedule

The schedule is the other integral part of a Halide program; it specifies how the algorithm computation is to be structured. This schedule entails specifying the storage allocation and computation of the stages of the pipeline in relation with each other. It also entails specifying how each stage is to be computed. The following shows a schedule for the 5x5 Gaussian blur filter:

```
op.compute_root()
    .parallel(y, 16)
    .vectorize(x, 128);
rows.compute_at(op, y);
```

This schedule specifies that the output stage (op) is computed by vectorizing the x dimension while computing the rows in the y dimension in parallel on multiple threads (each thread computes 16 rows of the output). Further, it says that the Func rows are computed in the `y` loop of op. The equivalent pseudo C code for this schedule is as follows:

```
for<parallel> (op.y = op.y.min; y < op.y.extent/16; ++op.y) { // parallel
loop over op.y
    // Compute the Func rows here, in the 'y' loop of op
    for (rows.y = rows.y.min; rows.y < rows.y.extent; ++rows.y) {
        for (rows.x = rows.x.min; rows.x < rows.x.extent; ++rows.x) {
            rows(rows.x, rows.y) = ...;
        }
    }

    for (op.y_inner = 0; y_inner < 16; ++op.y_inner) {
        for (op.x = op.x.min; op.x < (op.x.extent/128); ++op.x) {
            for (op.x_inner = op.x*128; op.x_inner < op.x*128 + 128; +
+op.x_inner) {
                op(op.x_inner, op.y_inner) = ...;
            }
        }
    }
}
```

All the other stages of the filter are not present by name in the pseudo code. By default, if a schedule is not specified for a Func, it is computed inline in its consumer. The following table describes some of the more important scheduling directives. For a more exhaustive list, take the tutorials hosted at https://halide-lang.org/tutorials/tutorial_introduction.html

Directive	Description
`.hexagon()`	In offload mode, indicates that the computation should be transitioned to the HVX processor. Upon encountering this directive, Halide automatically initiates the appropriate data transfers and the RPC calls to schedule the data transfers and computation on the HVX processor.
`reorder (dim_inner,…, dim_outer)`	Reorder to dimensions of a Func from *`dim_inner(most)`* to *`dim_outer(most)`*
`split(x, outer, inner, factor)`	Splits the dimension specified by *`x`* into inner and outer dimensions (specified by the variables *`inner`* and *`outer`*). The inner dimension iterates from *0* to *factor-1*.
`vectorizer(x)`	Vectorizes the dimension specified by *`x`*.
`unroll(x)`	Unrolls the dimension specified by *`x`*.
`fuse (inner, outer, fused)`	Fuses the dimensions specified by variables *`inner`* and *`outer`* into a single dimension specified by *`fused`*.
`parallel (x,size)`	Splits the dimension by size and parallelizes the outer dimension.
`title (x,y, xo, yo, xi, yi, xfactor, yfactor)`	Tiles the dimensions specified by the variables *`x`* and *`y`* by *`xfactor`* and *`yfactor`*. Each tile will be from *<xi, yi>* to *<xi+xfactor, yi+yfactor>*.

Directive	Description
`prefetch(func|img, var, [offset], [strategy])`	Prefetches the data accessed by *`func`* or *`img`* within the *`var`* loop body, accessing *`offset`* iterations beyond the current iteration using an optionally specified *`strategy`*.
`compute_at(consumer, dim)`	Specify *when* the producer stage is computed with respect to the consumer stage.
`compute_root()`	Place the computation of the stage at the top level (before anywhere it is used).
`store_at(consumer, dim)`	Specify *where* the memory for the producer stage is allocated with respect to the consumer stage.
`store_root()`	Place the allocation of the stage at the top level.
`align_storage(x, align)`	Aligns the dimension specified by *`x`* to be a multiple of the alignment specified by *`align`*.

hexagon

```
.hexagon()
```

Halide can offload parts of a pipeline to the Hexagon DSP with the `hexagon` scheduling directive. To enable the `hexagon` scheduling directive, include the `hvx_64` or `hvx_128` target features in the target. Use the HVX target features with an ARM Android host (to use Hexagon DSP hardware) or with an x86 host (to use the Hexagon simulator). The `hexagon` scheduling directive is supported only in the two offload modes – Device-offload and Simulator-offload.

reorder

```
.reorder(dim_inner, ..., dim_outer)
dim_inner: innermost dimension
dim_outer: outermost dimension
```

Reorders the dimensions of a Func.

If the dimensions of a Func are reordered, it translates to a reorder of the loops in the loop nest that computes the Func. For example, for the following algorithm, the equivalent C and C++ pseudo code are provided.

```
f(x, y) = input(x, y) + 10;
```

Pseudo code without reorder

```
for (f.y = f.y.min; f.y < f.y.extent; ++f.y) {
  for (f.x = f.x.min; f.x < f.x.extent; ++f.x) {
    f(f.x, f.y) = input(f.x, f.y) + 10;
  }
}
```

Pseudo code with reorder

```
f.reorder(x, y);
for (f.x = f.x.min; f.x < f.x.extent; ++f.x) {
  for (f.y = f.y.min; f.y < f.y.extent; ++f.y) {
    f(f.x, f.y) = input(f.x, f.y) + 10;
  }
}
```

split

```
.split(old_dim, new_outer_dim, new_inner_dim, factor, tail_strategy);
old_dim : dimension to be split
new_outer_dim: name of the new outer dimension
new_inner_dim: name of the new inner dimension.
factor       : factor to split old_dim by.
tail_strategy: strategy to handle tail or remainder loop if factor doesn't
divide old_dim completely.
```

Splits a dimension by a given factor into an inner and outer dimension.

The inner dimension iterates from 0 to (factor-1). This is useful because after the split, the inner and outer dimensions can be used with their own independent scheduling directives. For example, it is possible to vectorize the inner dimension while unrolling the outer dimension.

The following is the resulting C/C++ pseudo code for the following Halide code.

Halide:

```
f(x, y) = input(x, y) + 10;
f.split(x, x_o, x_i, factor);
```

C/C++ pseudo code:

```
for (f.y = f.y.min; f.y < f.y.extent; ++f.y) {
  for (f.x_o = f.x.min; f.x_o < (f.x.extent/factor); ++f.x_o) {
    for (f.x_i = 0; f.x_i < factor; ++f.x_i) {
      f.x = (f.x_o * factor) + f.x_i;
      f(f.x, f.y) = input(f.x, f.y) + 10;
    }
  }
}
```

vectorize

```
.vectorize(dim)
dim is computed all at once as a single vector.
.vectorize(dim, factor, tail_strategy)
dim           : Dimension to be split into an anonymous inner dimension
factor        : Factor by which dim should be split
tail_strategy : strategy to handle tail or remainder loop if factor doesn't
divide dim completely.
```

The vectorize directive has two variants:

- In the first variant, no split factor is specified and the entire dimension is vectorized.
- In the second variant, a split factor is specified by which the dimension is split and the anonymous inner dimension is vectorized entirely.

Thus, `vectorize` is a special case of split wherein the split factor also amounts to the vectorization factor.

unroll

```
.unroll(dim)
dim is unrolled completely.
.unroll(dim, factor, tail_strategy)
dim           : Dimension to be split into an anonymous inner dimension
factor        : Factor by which dim should be split
tail_strategy : strategy to handle tail or remainder loop if factor doesn't
divide dim completely.
```

The unroll directive has two variants:

- In the first variant, no split factor is specified and the entire dimension is unrolled.
- In the second variant, a split factor is specified by which the dimension is split and the anonymous inner dimension is unrolled entirely.

fuse

```
.fuse(inner, outer, fused)
inner: inner dimension
outer: outer dimension
fused: new fused dimension
```

Fuses the dimensions specified by variables x and y into a single dimension specified by fused. The new fused dimension covers the product of the inner and outer dimensions given.

parallel

```
.parallel(dim)
dim is computed inside a parallel loop.
.parallel(dim, task size, tail_strategy)
dim           : Dimension to be split into an anonymous inner dimension
task size     : The size of the inner dimension which is the size of each
                task
tail_strategy : strategy to handle tail or remainder loop if factor doesn't
divide dim completely.
```

The parallel directive has two variants:

- In the first variant, no split factor is specified and the entire loop for the dimension is run in parallel.
- In the second variant, a split factor is specified by which the dimension is split by the task size and the outer loop is executed in parallel.

Thus, `parallel` is a special case of split wherein the task size also amounts to the split factor.

tile

```
.tile(dim_0, dim_1, new_outer_dim¬_0, new_outer_dim_1, new_inner_dim¬_0,
new_inner_dim_1, factor_dim_0, factor_dim_1, tail_strategy);
dim_0          : dimension to be split (generally x)
dim_1          : dimension to be split (generally y)
new_outer_dim_0: name of the new outer dimension for dim_0
```

```
new_outer_dim_1: name of the new outer dimension for dim_1
new_inner_dim_0: name of the new inner dimension for dim_0
new_inner_dim_1: name of the new inner dimension for dim_1
factor_dim_0   : factor to split dim_0
factor_dim_1   : factor to split dim_1
tail_strategy  : strategy to handle tail or remainder loop if the
factors                 don't divide the extent of respective dimensions
```

Splits the two dimensions dim_0 and dim_1 and reorders them to create a tiled computation pattern for the stage. The affect is similar to the combination of the following:

1. Split `dim_0` into `new_outer_dim_0` (outer dim) and `new_inner_dim_0` (inner dim)
2. Split `dim_1` into `new_outer_dim_1` (outer dim) and `new_inner_dim_1` (inner dim)
3. Reorder them into `new_inner_dim_0`, `new_inner_dim_1`, `dim_0, dim_1`

This order effectively creates a tiled traversal over the image.

In another variant of tile directive, the old dimension is reused as the outer dimension. It effectively splits `dim_0` into `dim_0` (outer dim), and `new_inner_dim_0` (inner dim) and `dim_1` into `dim_1` (outer dim) and `new_inner_dim_1` (inner dim).

The following is the resulting C/C++ pseudo code for the following Halide code.

Halide:

```
f(x, y) = input(x, y) + 10;
f.tile(x, y, x_o, y_o, x_i, y_i, xfactor, yfactor);
```

C/C++ pseudo code:

```
for (f.y_o = f.y.min; f.y_o < f.y.extent/yfactor; ++f.y_o) {
  for (f.x_o = f.x.min; f.x_o < (f.x.extent/xfactor); ++f.x_o) {
      for (f.y_i = 0; f.y_i < yfactor; ++f.y_i) {
      f.y = (f.y_o * yfactor) + f.y_i;
      for (f.x_i = 0; f.x_i < factor; ++f.x_i) {
        f.x = (f.x_o * xfactor) + f.x_i;
        f(f.x, f.y) = input(f.x, f.y) + 10;
      }
    }
  }
}
```

prefetch

```
prefetch(func|img, var, [offset], [strategy])
func|img : Func or ImageParam to prefetch
var      : dimension to prefetch over
offset   : optional iteration offset
strategy : optional PrefetechBoundStrategy for generated code
```

The prefetch directive can be used to prefetch data for a `Func` or `ImageParam`.

This directive requests that data accessed by `func` or `img` within `var` loop body, accessing offset iterations beyond the current iteration that are prefetched into the L2 cache. If the `offset` is not

specified, it defaults to 1. If the prefetch `strategy` is not specified, the default is `PrefetchBoundStrategy::GuardWithIf`.

Available prefetch strategies include the following:

- `PrefetchBoundStrategy::GuardWithIf` – Guard the prefetch operation with an if statement to keep the prefetch within bounds.
- `PrefetchBoundStrategy::Clamp` – Clamp the computed bounds using min/max to keep the prefetch within bounds.
- `PrefetchBoundStrategy::NonFaulting` – Do not guard or clamp the region to prefetch.

This strategy assumes that prefetching out of bounds does not cause a fault. A fault can be avoided if out-of-bound reads do not cross a page boundary, or if the the memory buffer was allocated with sufficient padding to keep all accesses within allocated bounds.

Given the following Halide code fragment:

```
Func f;
Var x, y;
f(x, y) = input(x, y) * 2;
f.prefetch(input, y, 2);
```

The `input` buffer is prefetched within the y loop, preloading data with an offset of 2 iterations ahead. The resulting generated code looks similar to the following.

```
for y = 0, f.height
  if (y+2 < f.height)
    prefetch(&input[0, y+2], [0, input.width-1]);
  for x = 0, f.width
    f(x, y) = input(x, y) * 2;
```

compute_at

```
producer.compute_at(consumer, dim)
consumer: Any of the consumer stage for the current producer stage
dim     : dim inside which to compute the current stage
```

This directive is used to specify *when* the producer stage is computed with respect to the consumer stage. The allocation as well as computation for the producer function is adjusted to match the requirements for the computation of the consumer function. For example, consider the following pipeline:

```
g(x, y) = x*y;
f(x, y) = g(x, y) + g(x, y+1) + g(x+1, y) + g(x+1, y+1);
f.compute_root();
```

Pseudo code (C code equivalent) without `compute_at`

```
int f[height][width];
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    f[y][x] = x*y + x*(y+1) + (x+1)*y + (x+1)*(y+1);
}
}
```

In this case, `g` is said to have been computed inline.

Now, let's schedule `g` in the the `y` loop of `f`. This means, we'll place the computation of `g` in the `y` loop over `f`.

```
g(x, y) = x*y;
f(x, y) = g(x, y) + g(x, y+1) + g(x+1, y) + g(x+1, y+1);
g.compute_at(f, y);
```

Pseudo code with `compute_at`

```
int f[height][width];
for (int y = 0; y < height; y++) {
  int g[2][width+1];
  for (int x = 0; x < width; x++) {
    g[0][x] = x*y;
    g[1][x] = x*(y+1);
  }
  for (int x = 0; x < width; x++) {
    f[y][x] = g[0][x] + g[1][x] + g[0][x+1] + g[1][x+1];
  }
}
```

Note that only 2 rows for `g` are allocated now. This is because `g` is being computed in the `y` loop of `f`. In the `y` loop of `f`, we compute one row of `f`. This means that we need to only as much of `g` as is needed by one row of `f`. Each row of `f` depends on 2 rows of `g`. So, we only compute 2 rows of `g` in the `y` loop over `f`.

compute_root

```
.compute_root()
```

The `compute_root` directive is used to place the computation of the stage at the top level (before anywhere it is used). This directive avoids redundant computations of the same expressions in some pipelines at the cost of increased memory footprint and decreased locality. All of the independent functions are computed before any of the dependent functions. For example, consider the following pipeline:

```
g(x, y) = x*y;
f(x, y) = g(x, y) + g(x, y+1) + g(x+1, y) + g(x+1, y+1);
g.compute_root();
```

Pseudo code without `compute_root`

```
int f[height][width];
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    f[y][x] = x*y + x*(y+1) +
              (x+1)*y + (x+1)*(y+1);
  }
}
```

Pseudo code with `compute_root`

```
int f[height][width];
// Compute all of g before f
int g[height+1][width+1];
```

```
for (int y = 0; y < height+1; y++) {
  for (int x = 0; x < width+1; x++) {
    g[y][x] = x*y;
  }
}
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    f[y][x] = g[y][x] + g[y+1][x] + g[y][x+1] + g[y+1][x+1];
  }
}
```

Ensure to be careful using `compute_root` when offloading to Hexagon using the .hexagon scheduling directive. Remember, `compute_root` on a function means all of it is computed before the functions that need it are computed. For example, if `compute_root` is used as shown in the following pipeline, it can be seen that`g` gets computed on the host instead of Hexagon.

```
g(x, y) = x*y;
f(x, y) = g(x, y) + g(x, y+1) + g(x+1, y) + g(x+1, y+1);
f.hexagon();
g.compute_root();
```

The pseudo code (c code equivalent) looks like this:

```
int f[height][width];
// Compute all of g before f
// so, compute g before going to Hexagon
// to compute f
int g[height+1][width+1];
for (int y = 0; y < height+1; y++) {
for (int x = 0; x < width+1; x++) { g[y][x] = x*y;
}
}
// compute f on Hexagon.
for<Hexagon>(int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
f[y][x] = g[y][x] + g[y+1][x] + g[y][x+1] + g[y+1][x+1];
}
}
```

Since all of `g` is computed before `f`, `g` is computed on the host CPU.

store_at

```
producer.store_at(consumer, dim)
consumer: Any of the consumer stage for the current producer stage
dim     : dim inside which to allocate memory for the current stage
```

This directive is used to specify *where* the memory for the producer stage is allocated with respect to the consumer stage. The allocation size for the producer function is adjusted to match the

requirements for the computation of the consumer function. For example, consider the following pipeline:

```
g(x, y) = x*y;
f(x, y) = g(x, y) + g(x, y+1) + g(x+1, y) + g(x+1, y+1);
g.compute_at(f, x).store_at(f, y);
```

Pseudo code without `store_at` and `compute_at`

```
int f[height][width];
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    int g[2][2];
    g[0][0] = x*y;
    g[0][1] = (x+1)*y;
    g[1][0] = x*(y+1);
    g[1][1] = (x+1)*(y+1);
    f[y][x] = g[0][0] + g[1][0] + g[0][1] + g[1][1];
  }
}
```

Pseudo code with `store_at` and `compute_at`

```
int f[height][width];
for (int y = 0; y < height; y++) {
  int g[2][width+1];
  for (int x = 0; x < width; x++) {
    g[0][x] = x*y;
    g[0][x+1] = (x+1)*y;
    g[1][x] = x*(y+1);
    g[1][x+1] = (x+1)*(y+1);
    f[y][x] = g[0][x] + g[1][x] + g[0][x+1] + g[1][x+1];
  }
}
```

store_root

```
store_root()
```

This directive is used to place the allocation of the stage at the top level. Allocating storage outside the outermost loop can help avoid redundant computations of the same expressions at the cost of increased memory footprint. For example, consider the following pipeline:

```
g(x, y) = x*y;
f(x, y) = g(x, y) + g(x, y+1) + g(x+1, y) + g(x+1, y+1);
g.compute_at(f, y).store_root();
```

Pseudo code without `store_root` and `compute_at`

```
int f[height][width];
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    int g[2][2];
    g[0][0] = x*y;
```

```
    g[0][1] = (x+1)*y;
    g[1][0] = x*(y+1);
    g[1][1] = (x+1)*(y+1);
    f[y][x] = g[0][0] + g[1][0] + g[0][1] + g[1][1];
  }
}
```

Pseudo code with `store_root` and `compute_at`

```
int f[height][width];
int g[height+1][width+1];
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    if (x==0 || y==0)
      g[y][x] = x*y;
    if (y==0)
      g[y][x+1] = (x+1)*y;
    if (x==0)
      g[y+1][x] = x*(y+1);
    g[y+1][x+1] = (x+1)*(y+1);
    f[y][x] = g[y][x] + g[y][x+1] + g[y+1][x] + g[y+1][x+1];
  }
}
```

align_storage

```
.align_storage(x, align)
```

This directive pads the storage extent of a particular dimension of realizations of this function up to be a multiple of the specified alignment. This guarantees that the strides for the dimensions stored outside of dim will be multiples of the specified alignment, where the strides and alignment are measured in numbers of elements.

For example, to guarantee that a function `foo(x, y, c)` representing an image has scanlines starting on offsets aligned to multiples of 16, use `foo.align_storage(x, 16)`.

TailStrategy

Several scheduling directives, either explicitly or implicitly, split a dimension into two: an inner and an outer dimension. For example, `split` explicitly splits a dimension, while `vectorize` and `parallel` implicitly split a dimension. In terms of C/C++ code, this amounts to splitting the loop that traverses a dimension into two nested loops.

An important aspect of any such splitting is what should be done when the split factor doe s not fully divide the extent of the dimension. The so-called remainder loop can be handled in several different ways, and all such directives (such as `vectorize, parallel,` and `split`) take one last optional parameter called *TailStrategy*, which defines how the remainder loop should be handled. Following are the different types of TailStrategy:

- RoundUP: Rounds up the extent to be a multiple of the split factor. The benefit is that when vectorizing, Halide ensures that even the remainder loop is vectorized. The drawback, however, is that when this strategy is used to split the dimension of a stage that reads input or writes output, it assumes that the input or output image size is a multiple of the split factor. If the size is not a

multiple of the split factor, the image is accessed out of bounds. This case causes the application to fault unless the image allocation is sufficiently padded to allow for the out-of-bounds access.

- GuardWithIF: Guards the inner loop with an if condition and thereby ensures that Halide never evaluates beyond the extent of the dimension. This strategy is always legal and does not constrain the size of the input or the output. However, the drawback is that it inserts a conditional into the inner loop, and, in the remainder case, vectorization generates scalar code.
- ShiftInWards: Ensures that Halide does not evaluate beyond the extent of the original dimension by shifting the remainder case inwards. This strategy is always legal. If used on a stage that reads input or writes output, it only requires that the input or the output extent be at least the split factor.

  This strategy also supports vectorization, like `RoundUp`, because the inner loop will always be split-factor wide with no conditionals in it. The result is that some redundant values are computed near the end of the dimension.
- Auto: For pure definitions, this strategy implies using `ShiftInwards`. For pure Vars in update definitions, it implies using `RoundUp`, and for `RVars` in update definitions; it implies `GuardWithIf`.

5 High performance with Halide for HVX

Halide is unique in its separation of the algorithm from the schedule. Such a separation allows the programmer to freeze the algorithm and search through the schedules to improve performance. Guidelines for doing so are presented in the following sections.

NOTE The following strategies are for the advanced Halide user. Mastery of these strategies is not necessary to successfully author code with Halide for HVX.It is, however, recommended to follow the advice listed in this section to improve the performance of Halide pipelines.

5.1 Alignment

Align vector loads or stores for best performance. When large external buffers are allocated with the Hexagon device interface (`halide_hexagon_device_interface`), buffers are aligned to the natural vector width.

Align internal buffers

For internal pipeline stages, use `align_storage` to force alignment.

```
// Pad the storage extent of the 'x' dimension of the storage allocated
// for 'bounded_input' to be a multiple of 128. This ensures that the
// strides of dimensions outside 'x' are multiples of the specified alignment.
// Strides and alignment are viewed in terms of alignment here.
bounded_input
.compute_at(Func(output), y)
.align_storage(x, 128)
.vectorize(x, vector_size, TailStrategy::RoundUp);
```

Align external buffers

For ImageParams and outputs to have aligned loads and stores, use `set_host_alignment` and `set_stride`.

```
const int vector_size = get_target().has_feature(Target::HVX_128) ? 128 : 64;
Expr input_stride = input.dim(1).stride();
// Set the stride of dimension 1 (y dimension) to be a multiple of the native
// vector size.
input.dim(1).set_stride((input_stride/vector_size) * vector_size);
// Set the expected alignment of the host pointer in bytes.
input.set_host_alignment(vector_size);
output.set_host_alignment(vector_size);
```

```
// Set the stride of dimension 1 (y dimension) to be a multiple of the native
// vector size.
Expr output_stride = output.dim(1).stride();
output.dim(1).set_stride((output_stride/vector_size) * vector_size);
```

Align when splitting dimensions

Use `TailStrategy::RoundUp` for directives that split a dimension regardless of whether the split is explicit (`split`) or implicit (`vectorize`).

```
// vectorize splits the 'x' dimension into an inner dimension of size
// vector_size and an outer dimension. if the extent of the outer dimension
// is not a perfect multiple of 'vector_size', then the code below will
ensure
// that we round up to the next vector boundary. If, however, this is used on
// stage that reads from the input or writes to the output, it constrains the
// input or the output size to be a multiple of the split_factor (vector_size
in
// this case).
bounded_input
.compute_at(Func(output), y)
.align_storage(x, 128)
.vectorize(x, vector_size, TailStrategy::RoundUp);
```

5.2 Memory locality

Locality affects the latency of memory. Achieving good memory locality can significantly improve the performance of a program.

Tiling

Use tiling to improve locality in Halide as shown in the following 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);
// This is a producer-consumer relationship between 'f' and 'g' that is
// schedule as "Compute 'g' in tiles of 256x32 tiles and compute 'f' as
// required by tiles of 'g'.
g.tile(x, y, xi, yi, 256, 32, TailStrategy::RoundUp)
```

Line buffering

Two issues arise with tiling on HVX:

- Reasonably sized tiles are smaller than two vectors because of the large vector widths supported by HVX
- When tiling stencils, it is difficult for the producer functions to satisfy native vector requirements to avoid scalarization

The solution is to use line buffering to produce lines of the producers as required by lines of the consumer.

```
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);
```

5.3 Memory allocation and zero-copy buffers

Zero-copy buffers are memory that is visible to the host processor and the Hexagon. When working with zero-copy buffers, Halide does not perform a copy when switching access between the host and Hexagon processors.

`halide_buffer_t` descriptors

Before allocating memory in Halide, define descriptors to specify the shape of the buffers as shown in the following example. Memory is then allocated using the descriptor.

```
#include "pipeline.h"         // Generated Halide pipeline header
#include "HalideRuntime.h"
...
const int width  = atoi(argv[1]);                 // Image dimensions
const int height = atoi(argv[2]);
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};         // Input buffer
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;
halide_buffer_t output_buf = input_buf;  // Output buffer, same type as input

// Image shape, for each dimension: min index, extent, stride
halide_dimension_t in_dim[] = {{0, width, 1}, {0, height, stride}};
halide_dimension_t out_dim[]  = {{0, width, 1}, {0, height, stride}};

input_buf.dim  = in_dim;
output_buf.dim = out_dim;
```

Another option is to specify the Image shape using `halide_dimension_t` field names:

```
// Image shape, for each dimension: min index, extent, stride
halide_dimension_t in_dim[2] = {0};
halide_dimension_t out_dim[2] = {0};
```

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

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

input_buf.dim  = in_dim;
output_buf.dim = out_dim;
```

halide_device_malloc() / halide_device_free()

Use `halide_device_malloc/free` to manage zero-copy memory.

```
#include "HalideRuntimeHexagonHost.h"
...
// Allocate buffers
halide_device_malloc(nullptr, &input_buf, halide_hexagon_device_interface());
halide_device_malloc(nullptr, &output_buf,
halide_hexagon_device_interface()); if (input_buf.host == NULL ||
output_buf.host == NULL) {
printf("Error: Cannot allocate memory\n"); return 1;
}
...
// Free buffers halide_device_free(NULL, &input_buf);
halide_device_free(NULL, &output_buf);
```

rpcmem_alloc() / rpcmem_free()

If adding Halide to an existing application already using the SDK rpcmem library for allocating zero-copy buffers, then memory allocated with `rpcmem_alloc` can be attached to a `halide_buffer_t` using `halide_hexagon_wrap_device_handle()`.

```
#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); output_buf.host = (uint8_t*)rpcmem_alloc(25, RPCMEM_DEFAULT_FLAGS,
```

```
bufsize); if (input_buf.host == NULL || output_buf.host == NULL) {
printf("Error: Cannot allocate memory\n"); return 1;
}
halide_hexagon_wrap_device_handle(nullptr, &input_buf, input_buf.host,
bufsize);
halide_hexagon_wrap_device_handle(nullptr, &output_buf, output_buf.host,
bufsize);
...
// Free buffers rpcmem_free(input_buf.host); rpcmem_free(output_buf.host);
rpcmem_deinit();
```

Build the rpcmem library

Perform the `halide_hexagon_wrap_device_handle()` call to populate the device information in the `halide_buffer_t` structure.

When using the rpcmem library, also link in `libadsprpc.so` (if using the ADSP Hexagon) or `libcdsprpc.so` (if using the CDSP Hexagon) to get zero-copy behavior.

```
32-bit Android ARM host:
 L$HEXAGON_SDK_ROOT/libs/common/remote/ship/android_Release
 ladsprpc
 L$HEXAGON_SDK_ROOT/libs/common/remote/ship/android_Release
 lcdsprpc
64-bit Android ARM host:
 L$HEXAGON_SDK_ROOT/libs/common/remote/ship/android_Release_aarch64
-ladsprpc
 L$HEXAGON_SDK_ROOT/libs/common/remote/ship/android_Release_aarch64
-lcdsprpc
```

NOTE If `rpcmem.a` is not already built in the SDK, run the following:

```
   cd $HEXAGON_SDK_ROOT
. setup_sdk_env.source cd libs/common/rpcmem/
32-bit Android ARM host:
make BUILD_QEXES="" V=android_Release
64-bit Android ARM host:
make BUILD_QEXES="" V=android_Release_aarch64
```

Halide::Runtime::Buffer template class

Halide provides a higher level C++ buffer template class for memory allocation.

```
#include "HalideRuntimeHexagonHost.h"
#include "HalideBuffer.h"
...
const int W = 1920; const int H = 1080;
// Hexagon's device_malloc implementation will also set the host pointer if
// it is null, giving a zero copy buffer. Halide::Runtime::Buffer<uint8_t>
in(nullptr, W, H, 3); Halide::Runtime::Buffer<uint8_t> out(nullptr, W, H, 3);
in.device_malloc(halide_hexagon_device_interface());
```

```
out.device_malloc(halide_hexagon_device_interface());
...
printf("Running pipeline...\n");
double time = Halide::Tools::benchmark(iterations, 10, [&]() { int result =
pipeline(in, out);
if (result != 0) {
printf("pipeline failed! %d\n", result);
}
});
```

The Buffer template class can be passed directly to a Halide pipeline because the class automatically extracts the required `halide_buffer_t`.

```
Halide::Runtime::Buffer load_image() / save_image()
```

The buffer template class provides methods to load and store images, which can be used when writing shell applications.

NOTE When using a Buffer constructor that loads or allocates data, this allocation is not made using zero-copy memory. It must be explicitly copied between the host and Hexagon device.

```
#include "HalideRuntimeHexagonHost.h"
#include "HalideBuffer.h"
#include "halide_image_io.h"
...
// Allocate Buffers
Halide::Runtime::Buffer<uint16_t> input = load_image(argv[1]);
Halide::Runtime::Buffer<uint8_t> output(input.width(),
input.height()); if (input_buf.data() == NULL || output_buf.data()
== NULL) {
printf("Error: Cannot allocate memory\n");
return 1;
}
...
// Copy to device (since Buffer<> is not using zero-copy memory)
input.set_host_dirty();
output.set_host_dirty();
input.copy_to_device(halide_hexagon_device_interface());
output.copy_to_device(halide_hexagon_device_interface());
...
// Copy back to host (since Buffer<> is not using zero-copy memory)
output.copy_to_host();
save_image(output, "output.pgm");
```

5.4 VTCM memory and scatter-gather operations

Vector tightly coupled memory (VTCM) is fast vector memory available on Hexagon versions v65 and above. This section describes the use of this memory in the Halide pipelines. To use it in Halide, add Halide feature `hvx_65` feature to the target when running the generator. The two main use cases include:

- **Intermediate Funcs**: Since VTCM is a faster memory, faster performance can be achieved by storing small intermediate buffers in it. This must be done carefully as VTCM memory is not very large and is shared (256KB on SDM845). So, it is possible that some allocations may not fit in VTCM. In such cases, the following error message can be seen in logcat – `'halide_vtcm_malloc returned NULL'`. These issues are typically addressed by using smaller tile sizes. VTCM can be used for intermediate Funcs in the following manner - memory for the Func f is allocated in VTCM.

  Usage:

```
f(x, y) = 2*input(x, y);
output(x, y) = f(x, y) + 2;

f.compute_at(output, y)
               .store_in(MemoryType::VTCM)
               .vectorize(x, 64);
```

- **Scatter/gather operations**: All scatter/gather operations in HVX work only on VTCM memory.

  NOTE Scatter-gather instructions on Hexagon are only present for 16 bit and 32 bit data-types. To use these operations on 8 bit data-types, the programmer should explicitly add a stage to cast the appropriate buffers to 16 bit data types, perform the scatter-gathers and then cast back down to 8 bit type. For example, see the FAQ section.

  - Gather: Gathers are simply table lookups in Halide. For example,

```
f(x) = input(x) * 2;
g(x) = f(input(x));    ------- (1)
output(x) = g(x);
```

    This is a typical lookup operation in Halide. Here, f is the look up table (LUT), input represents the indices to be looked up in the table and output is the output buffer. To use gather operations to perform this, both the output and the LUT should be in VTCM memory. So, here f and g should be in VTCM memory.

```
f
.compute_at(output, Var::outermost())
.store_in(MemoryType::VTCM)
.vectorize(x, 64);
g
.compute_at(output, Var::outermost())
.store_in(MemoryType::VTCM)
.vectorize(x, 64);
```

- Scatters: Interchanging LHS and RHS for statement (1) generates a scatter. However, since the values for the output buffer might not have values defined for all indices, a pure definition stage is required and then use the update definition to achieve a scatter. For example,

```
f(x) = 0;
f(input(x)) = x*input(x);         -------(2)
output(x) = f(x);
```

  This is a typical scatter operation in Halide. The memory buffer for f should be present in VTCM memory to generate scatter operations for this. So for this example, add the following schedule

```
f
.allow_race_conditions()
.compute_at(output, Var::outermost())
.store_in(MemoryType::VTCM)
.vectorize(x, 64);
f
.update(0)
.vectorize(x, 64);
```

  Vectorizing an RDom is unsafe and so, the programmer must make sure that scatters should be used only when vectorizing an RDom is safe. This is conveyed by the scheduling directive `allow_race_conditions()`. Without this, Halide does not allow vectorization of an update stage. Here assuming input(x) is unique for each x, it can be ensured that no race conditions exist. If false, do not add `allow_race_conditions()` or try to vectorize the update stage.

- Scatters accumulates: A '+=' instead of '=' in statement (2) generates a scatter accumulate. A common use case for this is histogram generation. For example,

```
hist(x) = 0;         // Pure definition
hist(input(x)) +=1; // Update definition
output(x) = hist(x);
```

  This is a general histogram example in Halide. Buffer hist should be present in VTCM memory to generate scatter_acc operations for this. Add the following to the schedule:

```
f
.allow_race_conditions()
.compute_at(output, Var::outermost())
.store_in(MemoryType::VTCM)
.vectorize(x, 64);
f
.update(0)
.vectorize(x, 64);
```

Frequently asked questions (FAQs)

1. How to check if scatter/gather/scatter_acc instructions are being generated?

   To confirm whether scatter/gather/scatter_acc instructions are being generated do the following:

   - On the command line, do

```
$> export HL_DEBUG_CODEGEN=1
```

- Compile the Halide pipeline. This generates a debug log on stderr
- Search for gather/scatter/scatter_acc in the debug log.

2. Where to find complete examples for scatter/gather usage.

   For more examples, see the gather, scatter and histogram examples in Halide 2.2 (or later)

3. How to use scatter-gather operations for 8-bit data-types?

   Scatter/gather instructions on Hexagon are supported for 16 and 32 bit data-types only. To scatter/gather 8 bit datatypes, one needs to add a stage to cast the appropriate buffers to 16 bits, perform the scatter or gather and then cast back down to 8 bit type. This is shown in the following example.

   It is formed in such a way that the look up table (LUT) is composed of 16 bit elements. It is then gathered and cast the output back to 8 bits.

```
class Gather8Bit : public Generator <Gather8Bit> {
public:

  Input<Buffer<uint8_t>> input{ "input", 2};
  Output<Buffer<uint8_t>> output{ "output", 2};

  // Algorithm
  void generate() {
    // Cast, so that the elements of the LUT
    // are 16 bit elements.
    f(x) = cast<int16_t>(input(x) * 2);
    g(x) = f(input(x));
    output(x) = cast<uint8_t>(g(x));

  }
  void schedule() {
    f .compute_at(output, Var::outermost())
      .store_in(MemoryType::VTCM)
      .vectorize(x, 64);
    g .compute_at(output, Var::outermost())
      .store_in(MemoryType::VTCM)
      .vectorize(x, 64);
    output.hexagon()
         .vectorize(x, 128);
  }

private:
    Var x{ "x" }, y{ "y" };
    Func f{"f"}, g{"g"};

};
```

5.5 Power APIs

For increased control over the power and performance of an application, specify the power level before powering on HVX by requesting a specific performance mode or explicitly specifying individual performance parameters. There are two ways of specifying the power level:

- Use a power mode
- Set individual performance parameters

Both of these methods should be used in device-offload mode.

Specify power level by mode

```
API
#include "HalideRuntimeHexagonHost.h"
int halide_hexagon_set_performance_mode(void *user_context,
halide_hexagon_power_mode_t mode, bool dcvs_enable=true);
```

`user_context` – Ignored. It can also be NULL.

`mode`: Power level value similar to selecting a voltage corner as documented in the Hexagon SDK documentation.

`dcvs_enable`: Turn DCVS participation on or off. The default value is True.

Mode value	Equivalent voltage corner (See Hexagon SDK documentation)	Alternate name of mode
halide_hexagon_power_low	HAP_DCVS_VCORNER_SVS	halide_hexagon_power_svs
halide_hexagon_power_nominal	HAP_DCVS_VCORNER_NOM	
halide_hexagon_power_turbo	HAP_DCVS_VCORNER_TURBO	
halide_hexagon_power_default	HAP_DCVS_VCORNER_DISABLE	
halide_hexagon_power_low_plus	HAP_DCVS_VCORNER_SVSPLUS	halide_hexagon_power_svs_plus
halide_hexagon_power_low_2	HAP_DCVS_VCORNER_SVS2	halide_hexagon_power_svs2
halide_hexagon_power_nominal_plus	HAP_DCVS_VCORNER_NOMPLUS	

Specify power level by performance parameters

The alternative to the method above is to set individual performance parameters. However, using the`halide_hexagon_set_performance_mode` API is recommended over this approach. The code below shows sample usage of `halide_hexagon_set_performance`.

```
#include "HalideRuntimeHexagonHost.h"
halide_hexagon_power_t perf;
perf.set_mips = 1;
perf.mipsPerThread = 825;
perf.mipsTotal = 1650;
perf.set_bus_bw = 1;
```

```
perf.bwMegabytesPerSec = 18750;
perf.busbwUsagePercentage = 100;
perf.set_latency = 1;
perf.latency = 10;
halide_hexagon_set_performance (NULL , &perf );
```

Save power when application is idle

If the power mode is not specified or set to `halide_hexagon_power_default`, it defaults to a dynamic value that is typically even lower in power and performance than the Low mode.

To let the device run the clock settings at a lower power level when the application is idle:

1. Set the performance mode to the specified level.
2. Set the performance mode back to default when the program is done running on the HVX.

```
#include "pipeline_hvx128.h"
#include "HalideRuntimeHexagonHost.h"
…
// To avoid the cost of powering HVX on in each call of the pipeline,
// set performance mode to turbo and power HVX on once now.
halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_turbo);
halide_hexagon_power_hvx_on(NULL);
printf("Running pipeline...\n");

double time = benchmark(iterations, 10, [&]() {
     int result = pipeline(&in, &out);
     if (result != 0) {
  }
});

printf("pipeline failed! %d\n", result);

printf("Done, time: %g s\n", time);
    // We're done with HVX for now, power it off.
halide_hexagon_power_hvx_off(NULL);
    // Set performance mode back to the default for lower idle power usage
halide_hexagon_set_performance_mode(NULL, halide_hexagon_power_default);
```

5.6 Running an APK on device

In order to run a Halide pipeline on Hexagon from an APK, implement the following commands.

1. Run the following command:

```
adb pull /vendor/etc/public.libraries.txt
```

2. Update `public.libraries.txt` by adding `libhalide_hexagon_host.so` (This makes this library public to be used by app). For example,

```
adb push public.libraries.txt /vendor/etc/
adb push lib/cdsp/arm-64-android/libhalide_hexagon_host.so /vendor/lib64
```

   NOTE Tto make this library public, it should be in /vendor/lib64 and not under /system/lib64).

3. For Android P, run the following command

```
adb push lib/v60/libhalide_hexagon_remote_skel.so /vendor/lib64
adb reboot
adb root
```

   NOTE To make this library public, it should be in /vendor/lib64 and not under /system/lib/rfsa/adsp/

4. Run the app.

5.7 User controllable error exits

By default, when the Halide runtime detects an error it will not trigger a crash dump. An API is provided which allows the application to control which error conditions will dereference a NULL pointer on the target which enables collecting a crashdump. The list of available error codes can be found in `Halide/include/HalideRuntime.h` (see `enum halide_error_code_t`).

Example 1: Set 2 errors to be null deref'ed.

```
int err_list[] = {halide_error_code_out_of_memory,
halide_error_code_copy_to_host_failed, 0};
    (void) halide_hexagon_set_error_fault_mask(NULL, false, err_list);
```

Example 2: Enable all errors by setting 2nd arg to true, (3rd arg ignored in this case)

```
void) halide_hexagon_set_error_fault_mask(NULL, true, err_list);
```

5.8 Tracing of target mallocs

API to allow tracing of target mallocs callable from host enabled via `halide_hexagon_set_malloc_tracing(NULL, [0,1,2])`

0 => off

1 => start and end

2 => all mallocs (produces a large amount of output)

5.9 Set heap_mem grow size

API can set the heap_mem grow size by calling `heap_mem_grow_size`

The amount of memory to grow the heap when more space is needed can be specified by calling `halide_hexagon_mem_set_grow_size` before running a pipeline. For example, to grow the heap by at least 16MB, but no more than MAX_INT32 (2GB-1):

```
const long long int grow_min = 16*1024*1024;
const long long int grow_max = MAX_INT32;
    halide_hexagon_mem_set_grow_size(NULL, grow_min, grow_max);
```

The default is halide_hexagon_mem_set_grow_size(0x100000/2, MAX_UINT64). This is a min of 512KB and the largest allowed maximum.

5.10 Android P and SDM8150/SDM845

Running a Halide application Android P generally requires placing a copy of libhalide_hexagon_remote_skel.so into the same directory as the executable. Additionally, if the device requires use of a testsig.so, a copy needs to be placed in the same directory as the executable.

Typically, for testing purposes, copy these two files from `/system/lib/rfsa/adsp`.

Example:

```
cd /data/local/tmp/hexagon_benchmarks
cp /system/lib/rfsa/adsp/libhalide_hexagon_remote_skel.so .
cp /system/lib/rfsa/adsp/testsig*.so .
./process
```

5.11 Tools for signing Halide offload mode applications

Since Halide offload mode applications are host executables containing embedded Hexagon shared objects, several tools are provided to help extract these objects for signing. The method for signing the objects themselves is not described here as that depends on how the target device has been configured.

- If signing can be done during Halide compilation: Copy the `Halide/tools/hl_signnow` script (for example, to `~/halide/scripts/`) and replace the “cp $1 $2” step in the script with the method needed to sign a shared object.

  ```
  $ setenv HL_HEXAGON_CODE_SIGNER ~/halide/scripts/hl_signnow
  <build the app>
  hl_signnow: signing /tmp/hvx_unsignedy3t97n.so as /tmp/hvx_signedYkst0t.so
  hl_signnow: signing /tmp/hvx_unsignedFOHXXK.so as /tmp/hvx_signedCZksV1.so
  ```

- If signing must done separately from Halide compilation: Copy the `Halide/tools/hl_signsav` & `hl_signuse scripts` (e.g. to `~/halide/scripts/`)

  Two phase process:

  a. Select hl_signsav & build to save the shared objects

  b. Select hl_signuse & build to use the signed shared objects

  ```
  $ /bin/rm -rf /tmp/hl_sign_$USER
  $ setenv HL_HEXAGON_CODE_SIGNER ~/halide/scripts/hl_signsav

  <build the app>
  hl_signsav: saving /tmp/hvx_unsignedilzcjc.so as /tmp/hl_sign_$USER/
  ```

```
lib000.so
hl_signsav: saving /tmp/hvx_unsignedjAQVqB.so as /tmp/hl_sign_$USER/
lib001.so

$ ls /tmp/hl_sign_$USER
lib000.so  lib001.so

<sign the libraries>

$ setenv HL_HEXAGON_CODE_SIGNER ~/halide/scripts/hl_signuse

<build the app a second time>
hl_signuse: copying /tmp/hl_sign_$USER/lib000.so to /tmp/
hvx_signedSRbsjF.so
hl_signuse: copying /tmp/hl_sign_$USER/lib001.so to /tmp/
hvx_signedXLAWUD.so
```

Note that the `/tmp/hl_sign_$USER` directory will be empty after the second build step as the libraries are moved to a 'done' subdirectory as they are used. If it needs to be build again, move these objects back up one level.

The hl_signsav script can also be used to obtain the Hexagon shared objects if they are needed for any other reasons.

5.12 Profile Halide code

Profiling authored code allows developers to understand the performance bottlenecks in their programs. Halide provides two mechanisms to profile Halide executables:

- Halide profiling feature [device-offload mode & simulator offload mode]
- The Hexagon profiler tool on the Hexagon simulator
- Profiling using sysMon

Halide profiling feature

Modes supported in:

- Device-offload mode
- Simulator-offload mode

This Halide compiler provided profiler that measures the execution time spent in each pipeline in the Halide program. It is enabled by using the profiing feature when running a Halide generator. To use the profiling feature, add “-profile” to the target string when running the Halide generator. Perform the following steps to use the Halide profiler with the offload model on the simulator or a target device.

1. Add the **profile** feature to `HL_TARGET`:

   ```
   HL_TARGET=arm-32-android-hvx_128-profile
   ```

2. At the end of the main program (not the Halide generator), add a call to `halide_profiler_report(nullptr)`.

3. Rebuild the application.
4. If running on a target device, enter `adb logcat | grep halide` while running the application to view the generated profile.

   NOTE The Halide profiler cannot be used in standalone mode as the profiling thread does not exist when running in standalone.

The following is sample output produced by the profiler.

```
01-28 18:20:20.588 22634 22634 I halide : conv3x3a16_hvx_128
01-28 18:20:20.588 22634 22634 I halide : total time: 58.127136ms samples:216
runs:32 time/run:1.816473ms
01-28 18:20:20.588 22634 22634 I halide : average threads used: 0.578704
01-28 18:20:20.588 22634 22634 I halide : heap allocations: 0  peak heap
usage: 0 bytes
01-28 18:20:20.588 22634 22634 I halide : conv3x3:               1.164ms
(64%)   threads: 0.350
01-28 18:20:20.588 22634 22634 I halide : input_boundary:        0.651ms
(35%)   threads: 1.
```

Hexagon profiler tool on the Hexagon simulator

Modes supported:

- Simulator-standalone
- Simulator-offload

Hexagon tools are shipped with a profiling utility called hexagon-profiler.This utility gives provides instruction-level profiler information such as numbers of stalls, types of stalls and number of packet commits. It can be used when running Halide pipelines in simulator-standalone or simulator-offload mode.

To get instruction-level profile information, do the following in simulator-standalone mode:

1. Run the standalone executable with`--timing` and`--packet_analyze` options:

   ```
   hexagon-sim <hexagon executable (elf)> --memfill 0x0 --nullptr=2 --
   simulated_returnval  --timing --packet_analyze process.json -- "<command
   line flags for your hexagon executable(elf)>"
   ```

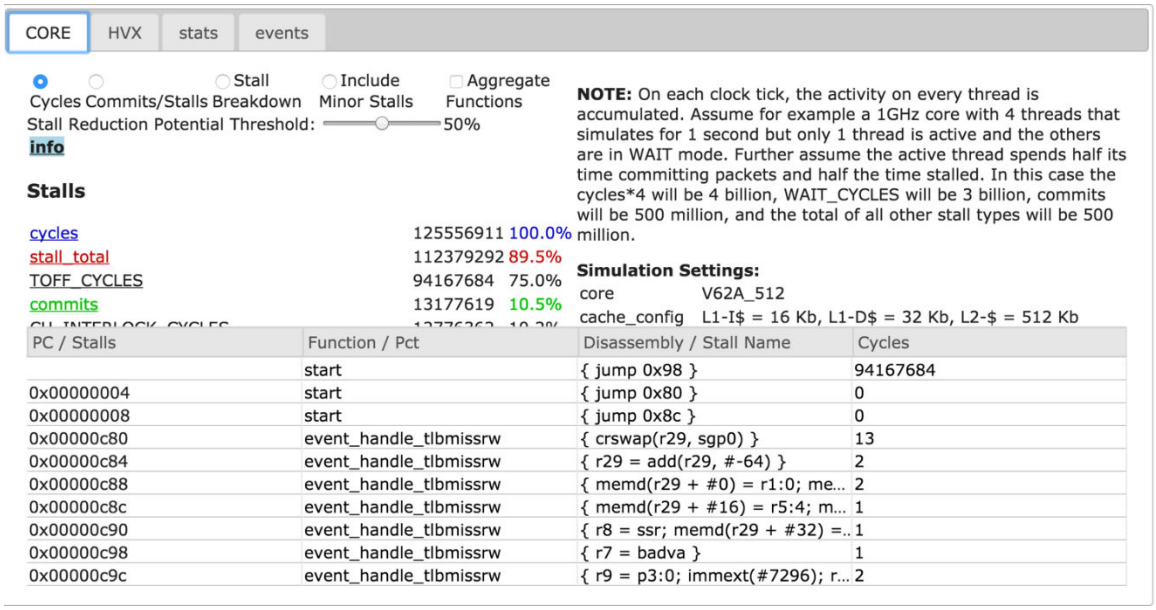
2. Run the Hexagon profiler on the generated statistics:

   ```
   hexagon-profiler --packet_analyze --json=process.json --elf=<your hexagon
   executable(elf)>  -o process.html
   ```

3. Open the generated HTML in a browser to view the profile

   ```
   open process.html
   ```

The following is a sample output produced by the Hexagon profiler.



4. Click the **HVX** tab to view the profile report for the vector instructions.

   For more information on using hexagon-profiler, refer to the Hexagon Profiler User Guide which is also available as part of the Hexagon_Document_Bundle distributed with the Hexagon Tools.

To get instruction-level profile information, do the following in simulator-offload mode:

1. Export `HL_HEXAGON_PACKET_ANALYZE=process.json`. Export `HL_HEXAGON_TIMING=1`
2. Ensure hexagon-profiler is in the path. Export `PATH=$PATH:<path-to-hex-tools>`.
3. Run the program normally in the simulator-offload mode. Each invocation of a halide pipeline emits a message similar to the following:

   ```
   To generate profile: hexagon-profiler --packet_analyze --json=process.json
   --elf=libhalide_shared_runtimeT15306325644329561485181.so:0x68000 -o
   process.html
   To generate profile: hexagon-profiler --packet_analyze --json=process.json
   --elf=libhalide_hexagon_codeT153063256274821212 7P5181.so:0x5f000 -o
   process.html
   ```

4. cd into the directory the program was running and should see a process.json file as well as a collection of halide pipelines libhalide*.so

   ```
   cd Halide/apps/blur/
   ```

5. Run one of the example commands that generates a process.html file that can be viewed with a browser.

   ```
   hexagon-profiler --packet_analyze --json=process.json --
   elf=libhalide_hexagon_codeT153063256274821212 7P5181.so:0x5f000 -o
   process.html
   ```

   For more information on using hexagon-profiler, refer to the Hexagon Profiler User Guide which is also available as part of the Hexagon_Document_Bundle distributed with the Hexagon Tools.

5.13 Profile Halide code with sysMon

Device-offload mode is supported for profiling Halide code with sysMon. It is also possible to profile code run on the Hexagon DSP using the sysMon profiler. For details on how to run sysMon, see the SDK documentation `${SDK_ROOT}/docs/Tools_sysMon_DSP_Profiler.html`

For finer granularity profiling, the target flag “hvx_sysmon” can also be added to the build target. This assigns a separate sysMon ID to each Halide function as well as the top level producer/consumer.

For example,

```
$ bin/camera_pipe_exec -o ./bin target=arm-64-android-hvx_128-hvx_sysmon -g
camera_pipe
...
Adding sysmon markers: max_depth(1)
HVX sysmon_id:1   offload_rpc.curved.s0.__outermost
HVX sysmon_id:2   produce:denoised
HVX sysmon_id:3   consume:denoised
```

To tag finer grain regions (at the expense of increased profiling overhead), set the HL_SYSMON_MAXDEPTH environment variable to control the depth to add sysmon_ids in the nested producer/consumers. The default setting is a depth of 1, so increasing the maximum depth to 2 yields more detail in the sysMon profile.

```
$ export HL_SYSMON_MAXDEPTH=2
$ bin/camera_pipe_exec -o ./bin target=arm-64-android-hvx_128-hvx_sysmon -g
camera_pipe
...
Adding sysmon markers: max_depth(2)
HVX sysmon_id:1   offload_rpc.curved.s0.__outermost
HVX sysmon_id:2   produce:denoised
HVX sysmon_id:3   produce:deinterleaved
HVX sysmon_id:4   consume:deinterleaved
HVX sysmon_id:5   consume:denoised
```

Alternatively, setting HL_SYSMON_MAXDEPTH to 0 avoids tagging any producer/consumers.

```
$ export HL_SYSMON_MAXDEPTH=0
$ bin/camera_pipe_exec -o ./bin target=arm-64-android-hvx_128-hvx_sysmon -g
camera_pipe
...
Adding sysmon markers: max_depth(0)
HVX sysmon_id:1   offload_rpc.curved.s0.__outermost
```

Increasing the maximum depth to 6 tags all producers & consumers in this example. Note that this also greatly impacts the execution time.

```
$ export HL_SYSMON_MAXDEPTH=6
$ bin/camera_pipe_exec -o ./bin target=arm-64-android-hvx_128-hvx_sysmon -g
camera_pipe
...
Adding sysmon markers: max_depth(4)
HVX sysmon_id:1   offload_rpc.curved.s0.__outermost
```

```
HVX sysmon_id:2   produce:denoised
HVX sysmon_id:3   consume:denoised
HVX sysmon_id:4   produce:deinterleaved
HVX sysmon_id:5   consume:deinterleaved
HVX sysmon_id:6   produce:g_b
HVX sysmon_id:7   consume:deinterleaved
HVX sysmon_id:8   produce:g_r
HVX sysmon_id:9   produce:output
HVX sysmon_id:10  produce:corrected
HVX sysmon_id:11  consume:corrected
HVX sysmon_id:12  consume:deinterleaved
HVX sysmon_id:13  consume:g_b
HVX sysmon_id:14  consume:g_r
```

The location of these identifiers in the generated code can be seen using HL_DEBUG_CODEGEN.

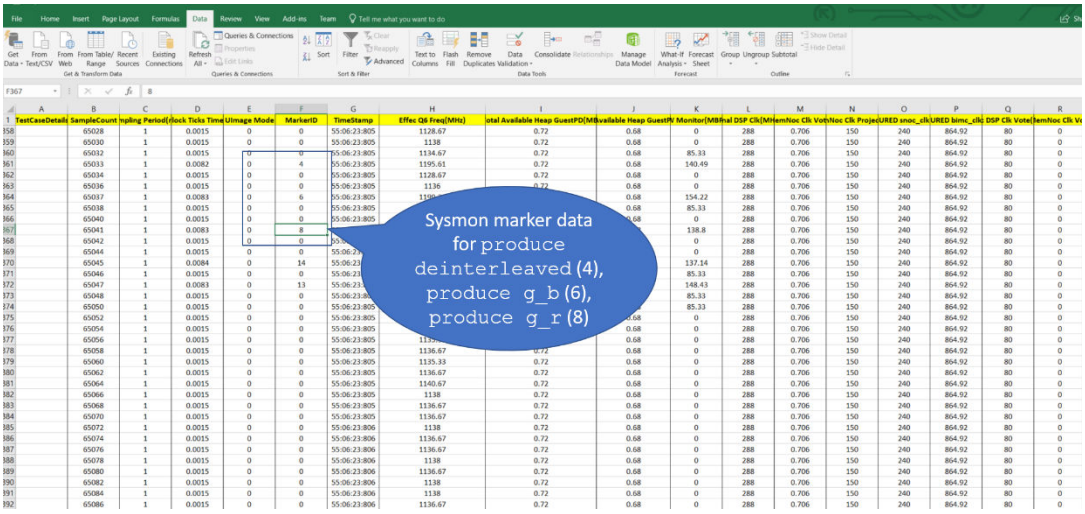
- On the command line, perform

  ```
  $ export HL_DEBUG_CODEGEN=1
  ```

- Compile the Halide pipeline. This generates a debug log on stderr
- Search for `halide_sysmon_marker` in the debug log.

```
produce deinterleaved {
      halide_sysmon_marker(4)
      consume denoised {
        halide_sysmon_marker(3)
            for (deinterleaved.s0.v1, deinterleaved.s0.v1.min_2,
(((curved.s0.v1.v3.base.s + curved.s0.v1.v3.v3) -
deinterleaved.s0.v1.min_2) + 2)) {
```

After running sysMonApp on the device, the following data can be received both on an Excel sheet and the Halide based executable concurrently.



A Halide with UBWCDMA

This addendum contains information about the addition of UBWCDMA function to Halide that is available on selected Snapdragon devices, with Compute DSP (cDSP) subsystem using Qualcomm Hexagon DSP core. It hi-lites the applicable Halide operatives that incorporates the UBWCDMA function to describe its usage, such as development requirements/example and use of scheduling directives.

Steps that are not hi-lited in this addendum, share the common steps described in corresponding sections of this document.

A.1 Introduction

UBWCDMA function is provided by an IP peripheral block on the cDSP subsystem, that has DMA capability for transferring data from DDR main memory to Hexagon local memory and in reverse direction, with the data (image frames) in DDR either compressed (bandwidth wise) or linear, for Hexagon DSP to process. The data in Hexagon local memory is un-compressed.

DMA direction	Description
DMA read	Transfer data from buffer in DDR main memory to buffer in Hexagon local memory
DMA write	Transfer data from buffer in Hexagon local memory to buffer in DDR main memory

With Halide, the UBWCDMA operations and interactions are largely abstracted, with the DMA driver calls hidden by the Hexagon UBWCDMA Runtime, and reduced to:

- A set of Host-side API calls to manage and specify DMA resources, and associate frame buffer to DMA, along with the buffer parameters
- A set of Halide scheduling directives to describe where, when, and how much DMA should be operating

For additional UBWCDMA information, refer to documentations included in Hexagon SDK.

Note that UBWCDMA is available on select Snapdragon devices, as identified below, as a hardware peripheral in the cDSP subsystem. And the term DMA and UBWCDMA can be used interchangeably to refer to the same function.

A.1.1 Support DMA formats

The list of DMA formats is shown in the following table, and its enumerations `(halide_hexagon_image_fmt_t)` defined in `$HALIDE_ROOT/Halide/include/HalideRuntimeHexagonDma.h`

Data format in DDR	Description
Raw	Unformatted and uncompressed, per byte DMA access
NV12	Linear, 8bpp semi-planar 4:2:0 YUV format, with each pixel component stored in 8b
NV12-4R	Same as linear NV12
P010	Linear, padded 10bpp semi-planar 4:2:0 YUV format, with each 10b pixel component represented in 16b; resulting in 2 bytes per pixel in Y plane and 4 bytes per pixel in UV plane
NV12 UBWC	Bandwidth compressed NV12
NV12-4R UBWC	Bandwidth compressed NV12-4R, encoded differently than NV12 compressed
P010 UBWC	Bandwidth compressed P010
TP10	Bandwidth compressed, tightly packed 10bpp semi-planar 4:2:0 YUV format; resulting in 4 bytes for every 3 pixels in Y plane and 8 bytes for every 3 pixels in UV plane

For Halide, DMA transaction is specified and scheduled separately for the Y vs. UV planes, due to the construct of the language, which requires the image frame to be represented by 2 separate Halide buffers; a Y and a UV buffer representation and must be described accordingly in its DMA-Buffer association (for example, use of NV12_Y, NV12_UV format for NV12 pixel format).

Note that an image formatted frame buffer must compose and contain either Y (luma) only, or both Y (luma) and UV (chroma) planes, per the format specification.

A.1.2 Supported Snapdragon device and Hexagon local memory

Table A-1

Snapdragon device	Hexagon local memory type	Size
SD8150	Hexagon's Locked L2 Cache	Upto 384KB*

NOTE *Provisioned by system

Note that Hexagon local memory is a shared and limited resource for the whole compute subsystem, whose available size is device specific, and its usage must be constraint with trade-off against required application performance.

A.1.3 Hexagon SDK compatibility

The minimum requirement is Hexagon SDK version 3.4.2.

A.1.4 Technical assistance

For questions about Halide for HVX/UBWCDMA, email halide@quicinc.com.

A.2 Getting started with Halide and UBWCDMA

The following examples describe building and running a Halide program with UBWCDMA.

- Build and run an example on the simulator

  Halide supports running Hexagon with UBWCDMA code on the simulator from the Hexagon tools, which uses the included UBWCDMA co-sim extension (for Snapdragon device with UBWCDMA support). Build and run the [dma] example on the simulator:

  ```
  cd $HALIDE_ROOT/Examples/ubwcdma/standalone/simulator/apps/[dma]
  ./test-dma
  ```

  Where the [dma] is

  - dmatest – basic DMA test using NV12 data (linear) format

- Build and run an example on a Snapdragon device

  Halide uses a small runtime library that must be present on the device.

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

  The libhalide_hexagon_remote_skel.so library must be signed or the device must be signed as a debug device to run Hexagon code. See `Hexagon_SDK/<sdk_rev>3.0/docs/Tools_Signing.html` for more information about signing Hexagon binaries.

  To build a Halide example – Halide/Examples/standalone/device/apps/[dma] – for execution on a Snapdragon device:

  a. Create a standalone toolchain from the Android NDK with the make-standalone- toolchain.sh script.

     ```
     export ANDROID_NDK_HOME=$HEXAGON_SDK_ROOT/tools/android-ndk-r14b/
     export ANDROID_ARM64_TOOLCHAIN=$ANDROID_NDK_HOME/install/android-21/
     arch- arm64
     $ANDROID_NDK_HOME/build/tools/make--standalone--toolchain.sh --
     arch=arm64
     …. --platform=android-21 --install-dir=$ANDROID_ARM64_TOOLCHAIN
     ```

  b. Run the example script provided.

     ```
     export RUNDROID=1
     export ANDROID_SERIAL=<device_id>
     cd $HALIDE_ROOT/Halide/Examples/standalone/device/apps/[dma]
     ./test-[dma]
     ```

Where [dma] is:

- dma_nv12_rw – basic DMA test using rawNV12 data (linear) format
- dma_raw_blur_rw – blur example with either NV12 (UBWC) or P010 (UBWC raw (8bit) data format

A.3 Programming with Halide and UBWCDMA

A.3.1 Halide for UBWCDMA runtime model

The Halide sources are compiled by the user to generate a binary, which is executed at runtime. Halide for UBWCDMA supports two runtime targets:

- Snapdragon devices
- Hexagon simulator

With Programmers support of generating Halide binaries for Standalone mode only:

- Standalone mode – The generated Halide binary is a standalone UBWCDMA object file. This file can be used to integrate Halide programs into an existing image processing pipeline.
  - On Snapdragon devices, the generated object file can be launched on the Hexagon processor. This runtime model is called the Device-standalone model.
  - On the Hexagon simulator runtime target, the generated object file can be used to simulate an end-to-end vision algorithm running on the Hexagon DSP, with UBWCDMA co-sim add-on. This runtime model is called the Simulator-standalone model.
- Offload mode – under future consideration, when support from Halide device interface class is extended

A.3.2 Halide device interface for UBWCDMA

These are UBWCDMA host-side APIs, for application code to set up and associate buffers to the DMA device for access to its services, as directed by the executing Halide pipeline scheduling directives.

These APIs are summarized below table, and the prototypes and detail descriptions are in:

`$HALIDE_ROOT/Halide/include/HalideRuntimeHexagonDma.h`

Table A-2

API name	Description
`halide_hexagon_dma_device_interface()`	For Halide::Runtime::Buffer's `device_wrap_native()` method to associate the buffer to UBWCDMA device
`halide_hexagon_dma_allocate_engine()`	Allocate a DMA engine
`halide_hexagon_dma_power_mode_voting()`	Set the DMA power mode, by voting for a pre-set performance level, as needed by the application. This power mode voting is separate from the Hexagon power votes

Table A-2 (cont.)

API name	Description
`halide_hexagon_dma_device_wrap_native()`	To attach buffer to DMA interface, when raw buffer (halide_buffer_t) is used
`halide_hexagon_dma_prepare_for_copy_to_host()`	Setup the DDR buffer specification for DMA read direction (transfer from DDR to local Hexagon memory), before any DMA transfers
`halide_hexagon_dma_prepare_for_copy_to_device()`	Setup the DDR buffer specification for DMA write direction (transfer from Hexagon local memory to DDR), before any DMA transfers
`halide_hexagon_dma_unprepare()`	Cleanup the buffer from DMA, after all transfers are completed
`halide_hexagon_dma_device_detach_native()`	To detach buffer from DMA interface, when raw buffer (halide_buffer_t) is used
`halide_hexagon_dma_deallocate_engine()`	De-allocate the DMA engine

Note that Hexagon local memory buffer is internal to the generated Halide pipeline that is managed by Halide UBWCDMA Runtime, and not visible to host-side user code. However, pipeline execution maybe impacted at runtime by its availability; see Memory locality for details.

A.3.3 Author and build your first Halide program

To start authoring code with Halide, modify an existing example for the Hexagon DSP simulator runtime target (Simulator-standalone mode). For instance, use an editor to open the following file:`$HALIDE_ROOT/Examples/ubwcdma/standalone/simulator/apps/[dma]_generator.cpp`

The examples are intended as templates to be modified as needed.

To build and run the Halide program on the simulator, invoke test-dma on Linux or test-dma.cmd on Windows. The compilation model in Halide for Hexagon DSP builds a Halide executable in two steps:

1. The native, x86 compiler is invoked on the Halide sources to build an x86 executable. This is called the generator.
2. The generator is executed to produce a Hexagon binary with UBWCDMA.

The test-dma script runs both these steps to produce a Hexagon DSP binary and then invokes the Hexagon simulator on the binary.

Where [dma] can be:

- dmatest – basic DMA test using NV12 data (linear) format

A.4 Introduction to the Halide programming language

A.4.1 Schedule with UBWCDMA

The following table lists scheduling directives that are important to working with UBWCDMA. Refer to http://halide-lang.org/docs/index.html for the description and construct of these directives, and DMA example code that illustrates its usage in`$HALIDE_ROOT/Examples/ubwcdma/standalone/*`.

Directive	Description
`copy_to_host()`	DMA data transfer from DDR to Hexagon local memory buffer (aka DMA read)
`copy_to_device()`	DMA data transfer from Hexagon local memory to DDR buffer (aka DMA write)
`store_in()`	Specify memory allocation type as Hexagon local memory, as required by DMA
`store_at()`	Place the allocation of Hexagon local memory buffer at a specific stage/loop (ie. Must be outside of inner-loops)
`reorder_storage()`	De-interleave the UV semi-plane for 4:2:0 pixel format in Hexagon local memory buffer
`tile()`	Defines the ROI dimensions that is factored into calculating the required Hexagon local memory buffer size
`bound()`	Bound DMA access to within the image frame buffer boundary for stencil processing
`fold_storage()`	Circular buffer of Hexagon local memory for async and/or stencil processing; to allow access in a monotonically increasing or decreasing order, by rotating the buffer. Also factored into the required Hexagon local memory buffer size
`async()`	Async control flow between the consumer and producer of fold_storage to allow concurrency of processing already transferred data on one fold while simultaneously DMA transferring data on another fold. That is, to prefetch next needed consumer data from DDR with DMA read, or to store resultant producer data to DDR with DMA write, while in parallel, processing (consuming or producing) another set of data
`split()`	Splits dimension of a frame buffer, precursor to independent DMA access per outer dimension split when used with parallel() directive
`parallel()`	Parallelize each of the split outer dimension, with independent assigned DMA access; to allow for parallel processing and DMA of data for each split
`compute_at()`	Specify when the producer stage is computed with respect to the consumer stage.
`compute_root()`	Place the computation of the stage at the top level (before anywhere it is used).
`store_at()`	Specify where the memory for the producer stage is allocated with respect to the consumer stage

Directive	Description
`store_root()`	Place the allocation of the stage at the top level.
`align_storage()`	Aligns the dimension specified by x to be a multiple of the alignment specified by align.

A.4.2 De-interleaving of UV for 4:2:0

For 4:2:0 semi-planar format, the UV-plane is expected to be after the Y-plane when stored within a frame buffer. The Y and UV planes are then cropped appropriately and presented as 2 individual Halide buffers to the Halide pipeline; with Y buffer described as 2D, and UV buffer described as 3D with the UV interleaved.

Refer to the NV12 example for proper coding to describe the UV interleaved property to Halide.
`$HALIDE_ROOT/Examples/dma_nv12_rw/dma_nv12_rw_generator.cpp`

A.4.3 Known limitations

Table A-3

Limitation	Description
`fold_storage()`	See limitation below
`async()`	Requires used of fold_storage() to ping-pong asynchronously between producer and consumer threads.
`Stencil`	Halide handling of stencil with `fold_storage()` is not working properly, and the problem is magnified by adding UBWCDMA. For stencil (N-points 2D) style processing, Halide/ UBWCDMA only works in linear data space and without folding. The observation is that the ROI size changes around the image frame edges to accommodate the out-of-bound pixels, which requires the per-pixel DMA capability that is only supported for linear data transfer.

A.5 High performance with Halide for UBWCDMA

A.5.1 Memory locality

With DMA data transfer into Hexagon local memory, it increases the transfer efficiency and reduces the access latency by the Hexagon DSP core, and thus improves the overall program's performance. The requirement of Hexagon local memory is mandatory for DMA, as either the destination memory type for DMA read from DDR, or the source memory type for DMA write to DDR.

The allocation of Hexagon local memory by the Halide pipeline is directed by `.store_in(MemoryType::LockedCache)` scheduling directive.

For each data buffer provided to Halide pipeline, there is at least one corresponding instance of Hexagon local memory buffer allocated to service it. The size of this local buffer is determined by several factors, including Halide tile size, value size, number of fold storage; roughly represented by:

```
local_size = align(tile_width x tile_height x sizeof(data), 128) x
number_of_folds
```

The number of allocated local memory instances per Halide data buffer is determined by parallelizable aspect of Halide directives, such as number of splits for multi-threading; roughly represented by number of DMA threads, which should be limited to number of Hexagon DSP cores, to avoid un-necessary resource allocation:

```
number_of_DMA_threads = max(number_of_splits, number_of_hexagon_cores)
```

Note that when the supported local memory is allocated from L2 cache (Snapdragon device specific), which is a limited resource, correspondingly reduces the amount of available L2 cache memory by locking portion of it. It also has possible affects on the overall processing performance due to reduced cache availability. Thus, increasing the transfer size increases the DMA efficiency, while reduces the amount of resource available to service other processing needs.

A.5.2 Power APIs

The level of DMA performance is controlled by its power modes, which translates into a corresponding set of pre-configured performance parameters. Each Snapdragon device has a specific table of performance parameters that is 1:1 mapped to the supported power modes. The application determines its performance requirement and votes for the power mode accordingly, via the power API, for both the Hexagon DSP core and DMA; DMA performance level is voted independently from the Hexagon DSP performance level.

The power mode must be voted after a DMA engine is allocated and can re-vote any time during its allocation life time. If there is no power mode vote, then the default power mode is applied. On application exit and before de-allocating the DMA engine, un-vote the power mode back to default mode.

All enumerated power modes are supported by the API. If a specific Snapdragon device does not have support for a mode, that requested power mode is mapped to the closest (next or higher) appropriate supported mode.

DMA power modes are enumerated by `halide_hexagon_power_mode_t`

```
#include "HalideRuntimeHexagonHost.h"
#include "HalideRuntimeHexagonDma.h"

halide_hexagon_dma_power_mode_voting(NULL, halide_hexagon_power_low_2);
halide_hexagon_dma_power_mode_voting(NULL, halide_hexagon_power_low);
halide_hexagon_dma_power_mode_voting(NULL, halide_hexagon_power_low_plus);
halide_hexagon_dma_power_mode_voting(NULL, halide_hexagon_power_nominal);
halide_hexagon_dma_power_mode_voting(NULL, halide_hexagon_power_nominal_plus);
halide_hexagon_dma_power_mode_voting(NULL, halide_hexagon_power_turbo);
halide_hexagon_dma_power_mode_voting(NULL, halide_hexagon_power_default);
```

Example DMA power mode performance mapping for SD8150 device:

Halide power mode	Voltage corner	Frequency*	Bandwidth**
halide_hexagon_power_low_2	SVS2	131250000	262500000
halide_hexagon_power_low	SVS	245600000	491200000
halide_hexagon_power_low_plus	SVS L1	262500000	525000000
halide_hexagon_power_nominal	Normal	409330000	818660000
halide_hexagon_power_nominal_plus	Normal		
halide_hexagon_power_turbo	Turbo	525000000	1050000000
halide_hexagon_power_default	SVS		

A.5.3 DMA concurrency

Pointers for concurrency considerations to improve performance:

- Split frame for parallel processing to take advantage of multi-threaded Hexagon DSP core
  - Processing for each split is scheduled into a separate HW thread
  - Best practice is for number of splits, less than or equal to the number of supported HW thread (Snapdragon device dependent)
  - Number of splits more than available HW threads incurs un-necessary overheads for the ones above the number of supported HW threads; such as thread scheduling including context switching, and any associated cycles for resource allocation/de-allocations
- Async between Halide producer and consumer to allow simultaneous DMA data transfer and data processing to compress processing timeline
  - DMA is a HW accelerator that runs independent of Hexagon DSP core for the data transfer
  - This method improves performance by removing DMA transfer waits with ahead of time data transfer, such that data is already available when Hexagon DSP core needs it
  - Best practice is to transfer right sized data that takes longer to process by the Hexagon DSP core than to DMA transfer it
  - Best practice is to apply ping-pong buffers, using fold_storage(), to enable this async parallelism between producer and consumer. This number of folds is dictated by the implemented algorithm of how much data ahead (future) and behind (history) is required
  - Refer to Memory locality for local memory constraints, for right sizing the data buffer

- Deploying multiple independent DMA engines to allow for simultaneous transfer of data, to potentially shorten the overall timeline by reducing the data transfer time
  - Logistically the per application DMA engine separation can be:
    - Per frame buffer DMA engine
    - Read vs. write DMA engine
  - Points to consider for the potential timeline saving
    - Limited number of available DMA engines, with allocated engines only shared within application and not across applications
    - Transfer unit is sized small (local memory resource constraint) such that the overall difference between serialization vs. parallelization of transfer is minute
    - Asynchronous ping-pong style DMA transfer maybe a better option

B References

Title	Number
Resources	
Halide programming language	http://halide-lang.org/
Halide programming tutorials	http://halide-lang.org/tutorials/tutorial_introduction.html
Halide Language Documentation	http://halide-lang.org/docs/index.html

Glossary

Term	Definition
HVX	Hexagon Vector eXtensions