Qualcomm Technologies, Inc.

Halide for HVX

User Guide

80-PD002-1 Rev. B

October 20, 2017

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.

Contents

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 the Hexagon HVX
- As runtime libraries to support running code on the Hexagon HVX

Halide for Hexagon Vector eXtensions (HVX) allows developers to utilize the features of the HVX processor without knowledge of the underlying HVX architecture. It also supports offloading of work 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

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 $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/3.0/docs/Tools_Signing.html for more information about signing Hexagon binaries.

To build a Halide example – the sobel filter – for execution on a Snapdragon device:

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.

   ```
   cd $HALIDE_ROOT/Halide/Examples/sobel
   ./test-sobel
   ```

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

```
setup-env.cmd
cd %HALIDE_ROOT%\Examples\sobel test-sobel.cmd
```

3 Programming with Halide for HVX

3.1 Halide for HVX Runtime Model

The Halide sources 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

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. This runtime model is aclled the Simulator-standalone model.

3.2 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:

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.

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:

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

This schedule specifies that the output stage 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 output. The equivalent pseudo C code for this schedule is as follows:

```
for<parallel> (output.y = output.y.min; y < output.y.extent/16; ++output.y)
{ // parallel loop over output.y
    // Compute the Func rows here, in the 'y' loop of output
    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 (output.y_inner = 0; y_inner < 16; ++output.y_inner) {
        for (output.x = output.x.min; output.x < (output.x.extent/128); +
+output.x) {
            for (output.x_inner = output.x*128; output.x_inner < output.x*128
+ 128; ++output.x_inner) {
                output(output.x_inner, output.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>`*.
`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`*.

Directive	Description
`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 will be 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 will 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 will look similar to this:

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

Pseudo code without `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][0for (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) {] = 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 `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];
  }
}
```

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. 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++) {
    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 `compute_root`

```
int f[height][width];
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];
  }
}
```

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.

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 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.

Specify power level by mode

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

Specify power level by performance parameters

```
#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);
```

5.5 Profile Halide code

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

- Integrated Halide profiler
- The Hexagon profiler tool on the Hexagon simulator

The integrated Halide profiler

This Halide-provided profiler measures the execution time spent in each pipeline in the Halide program. 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 your 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

To get instruction-level profile information, perform the following steps to use the Hexagon profiler tool on Halide programs run in standalone mode on the simulator.

NOTE This profiling mechanism only works for Halide programs running in standalone simulation mode.

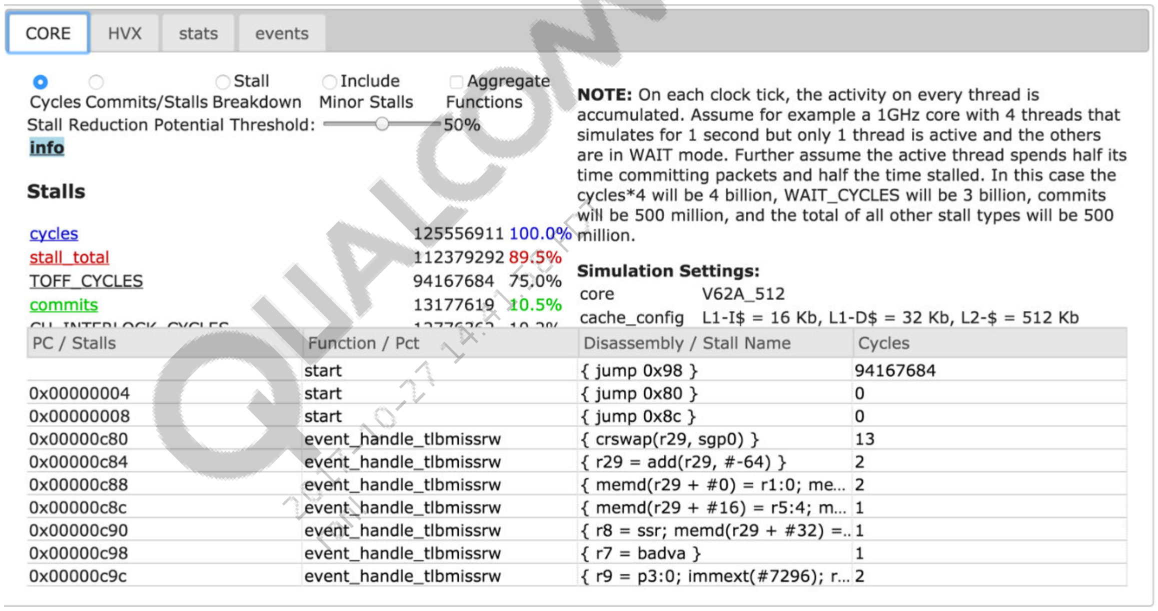
1. Run the standalone executable with `--timing` and `--packet_analyze` options:

```
env ARCHSTRING="--l2cache_perfect 1" hexagon-sim process --memfill 0x0 --
nullptr=2 --simulated_returnval  --timing --packet_analyze process.json --
"<options to process executable>"
```

2. Run the Hexagon profiler on the generated statistics:

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

3. Open the generated HTML in a browser to view the profile, `firefox process.html`. The following is a sample output produced by the Hexagon profiler.



CORE
HVX
stats
events
[✓ Cycles] Commits/Stalls Breakdown
Stall Reduction Potential
Include Minor Stalls
Aggregate Functions
Stall Reduction Potential Threshold: 50%
info

Stalls

cycles	125556911	100.0%
stall_total	112379292	89.5%
TOFF_CYCLES	94167684	75.0%
commits	13177619	10.5%

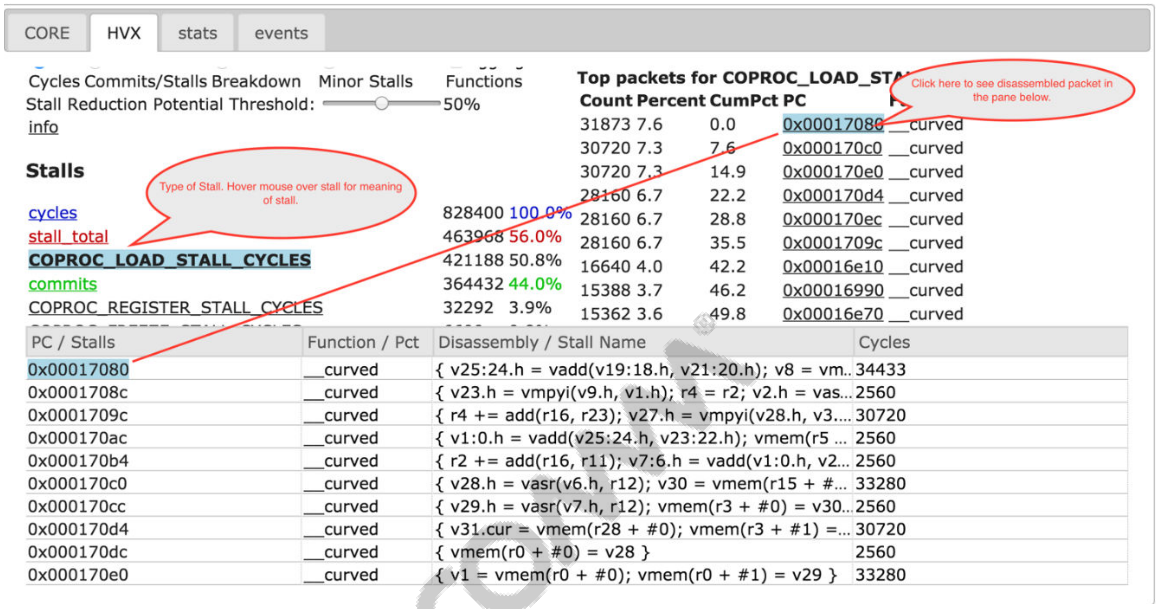
NOTE: On each clock tick, the activity on every thread is accumulated. Assume for example a 1GHz core with 4 threads that simulates for 1 second but only 1 thread is active and the others are in WAIT mode. Further assume the active thread spends half its time committing packets and half the time stalled.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

Simulation Settings:

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

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

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


CORE
HVX
stats
events
Click here to see disassembled packet in the pane below.
Type of Stall. Hover mouse over stall for meaning of stall.

Cycles Commits/Stalls Breakdown Minor Stalls Functions
Stall Reduction Potential Threshold: 50%
info

Stalls

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

Top packets for COPROC_LOAD_STA

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

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

A 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