Halide for Hexagon™ DSP with Hexagon Vector eXtensions (HVX) using LLVM

Pranav Bhandarkar, Anshuman Dasgupta, Ron Lieberman, Dan Palermo
Qualcomm Innovation Center (QuIC)
Dillon Sharlet, Andrew Adams (Google)
4th Feb 2017
Agenda

1. Halide
2. Hexagon with HVX
3. Implementation details of the Halide Compiler
4. Example 1: blur5
5. Example 2: camera_pipe
Halide
A new DSL for image processing and computational photography.

- Fast image-processing pipelines are difficult to write.
  - Definition of the stages of the pipeline.
  - Optimization of the pipeline - vectorization, multi-threading, tiling, etc.

- Traditional languages make expression of parallelism, tiling and other optimizations difficult to express.

- Solution: Halide enables rapid authoring and evaluation of optimized pipelines by separating the algorithm from the computational organization of the different stages of the pipeline (schedule).

- Programmer defines both, the algorithm and the schedule.

- Front end embedded in C++.

- Compiler targets include x86/SSE, ARM v7/NEON, CUDA, Hexagon™/HVX and OpenCL.
Halide
A new DSL for image processing and computational photography.

- Halide programs / pipelines consist of two major components
  - Algorithm
  - Schedule

- Algorithm specifies what will be computed at a pixel.
- Schedules specifies how the computation will be organized.

```cpp
ImageParam input(Uint(8), 2) // Image with 8 bits per pixel.
Halide::Func f;

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

// Schedule
f.vectorize(x, 128).parallel(y, 16);
```
Agenda

1. Halide
2. Hexagon with HVX
3. Implementation details of the Halide Compiler
4. Example 1: blur5
5. Example 2: camera_pipe
Hexagon™ Processor

- 32 bit VLIW Processor.
- “Packets” group 1 to 4 instructions for parallel execution.
  - Compiler / assembly coder chooses instructions for parallel execution; No NOP padding necessary.
- 4 Hardware threads.
- FFT and circular addressing modes.
- Native numerical support for fractional real+imaginary data.
- Modern system architecture with precise exceptions, MMU with address translation and protection and capable of support Linux, Real-Time OS, etc.
Hexagon V60 with HVX

- Large vector (SIMD) extensions
  - 2 1024b vector contexts configurable as 4 512b vector contexts as well.
- Vectors can hold 8-bit bytes, 16-bit halfwords, or 32-bit words.
- L2 is the first level memory for vector units.
Halide & HVX

Motivation

Performance of the LLVM Compiler on Hexagon V60 with HVX using C with intrinsics.

Normalized hand coded assembly = 1. Higher is better.
Halide & HVX

Motivation

Performance of the LLVM Compiler on Hexagon V60 with HVX using C with intrinsics.

Normalized hand coded assembly = 1. Higher is better.

• Intrinsics are good, but require low level architecture knowledge to extract performance.

• How do we leverage the LLVM compiler with a good HVX backend?
Agenda

1. Halide
2. Hexagon with HVX
3. Implementation details of the Halide Compiler
4. Example 1: blur5
5. Example 2: camera_pipe
Halide Compiler
Ahead-of-time (AOT) compilation.
Halide on Hexagon with HVX

- Halide provides two execution environments for HVX.

- Hardware model or the offload model.
  - Transparently dispatches Halide pipeline from the host CPU to the Hexagon™ processor.
  - Very easy to use as a developer.

```cpp
ImageParam input(Uint(8), 2) // Image with 8 bits per pixel.
Halide::Func f;

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

- Standalone model, which can be used for both on-device execution and simulation.
  - Simpler startup.
  - Allows us to prototype future hardware features.
Halide on Hexagon with HVX
Vectorization, Alignment & Prefetching.

• HVX supports unaligned loads, but they are less efficient than aligned loads.
• Halide provides an abstraction to specify assumptions about the alignment of external memory buffers.
• Halide also provides a scheduling directive to prefetch data into the L2 cache. For example, “my_func.prefetch(y, 2)” will prefetch into the L2 cache, 2 iterations worth of data needed in the ‘y’ loop.
• HVX vectors are deinterleaved when widened and interleaved back together on truncation. Halide keeps track of “lanes”
• HVX supports unaligned loads, but they are less efficient than aligned loads.
• Halide provides an abstraction to specify assumptions about the alignment of external memory buffers.
• Halide also provides a scheduling directive to prefetch data into the L2 cache. For example, “my_func.prefetch(y, 2)” will prefetch into the L2 cache, 2 iterations worth of data needed in the ‘y’ loop.

HVX vectors are deinterleaved when widened and interleaved back together upon truncation. Halide keeps track of “lanes”

\[ \text{Deinterleaved double vector of words} \]
\[
\text{Interleaved vector of half words}
\]

**Halide**

\[
i16\_sat(i32(i16_1))
\]

**HVX**

\[
v0.h=vrsat(v1.w, v2.w)
\]

\[
i16\_sat(i32_1)
\]

\[
v0.h=vpack(v1.w, v2.w)\_sat
\]
Halide & LLVM (Median* Filter)

```c
.falign
.LBB135_11:
  {
    v5=valign(v10,v5,#1)
v29.ub=vmax(v9.ub,v8.ub)
v18=vmem(r1++#1)
    vmem(r0++#2)=v15
  }
  {
    v28.ub=vmin(v5.ub,v11.ub)
v16.ub=vmin(v25.ub,v20.ub)
v12.ub=vmax(v13.ub,v12.ub)
    v25.cur=vmem(r20++#1)
  }
  {
    v9.ub=vmax(v25.ub,v20.ub)
v19.ub=vmax(v25.ub,v20.ub)
v21.ub=vmin(v16.ub,v18.ub)
    v13=vmem(r10++#1)
  }
  {
    v11.ub=vmin(v9.ub,v18.ub)
v5.ub=vmax(v9.ub,v18.ub)
v20.ub=vmin(v25.ub,v20.ub)
    v10=vmem(r20++#1)
  }

* The assembly for the entire inner loop is not shown here.
```
Halide & LLVM (Median* Filter)

* The assembly for the entire inner loop is not shown here.
Agenda

1. Halide
2. Hexagon with HVX
3. Implementation details of the Halide Compiler
4. Example 1: blur5
5. Example 2: camera_pipe
Gaussian 5 point blur

Halide Code

1. // Define a 1D Gaussian blur (a [1 4 6 4 1] filter) of 5 elements.
2. Expr blur5(Expr x0, Expr x1, Expr x2, Expr x3, Expr x4) {
3.     // Widen to 16 bits, so we don't overflow while computing the stencil.
4.     x0 = cast<uint16_t>(x0);
5.     x1 = cast<uint16_t>(x1);
6.     x2 = cast<uint16_t>(x2);
7.     x3 = cast<uint16_t>(x3);
8.     x4 = cast<uint16_t>(x4);
9.     return cast<uint8_t>((x0 + 4*x1 + 6*x2 + 4*x3 + x4 + 8)/16);
10. }
11. // Algorithm
12. ImageParam input(UInt(8), 3);
13. // Apply a boundary condition to the input.
14. Func input_bounded("input_bounded");
15. input_bounded(x, y, c) = BoundaryConditions::repeat_edge(input)(x, y, c);
16. // Implement this as a separable blur in y followed by x.
17. Func blur_y("blur_y"), blur("blur");
18. blur_y(x, y, c) = blur5(input_bounded(x, y - 2, c), input_bounded(x, y - 1, c), input_bounded(x, y, c), input_bounded(x, y + 1, c), input_bounded(x, y + 2, c));
19. blur(x, y, c) = blur5(blur_y(x - 2, y, c), blur_y(x - 1, y, c), blur_y(x, y, c), blur_y(x + 1, y, c), blur_y(x + 2, y, c));
Gaussian 5 point blur
Halide: Schedule 1 - Vectorize

vector_size = 128;
blur.compute_root().hexagon().vectorize(x, vector_size);

Loop Nest:
produce blur:
  for __outermost in [0, 0]<Hexagon>:
    for c:
      for y:
        for x.x:
          vectorized x.tmp in [0, 127]:
            blur(...) = ...

Run on device:
Using HVX 128 schedule
Running pipeline...
Done, time: 0.0483019 s
Success!
Gaussian 5 point blur

Halide: Schedule 2 - compute_root

```cpp
vector_size = 128;
input_bounded.compute_root();
blur.compute_root().hexagon().vectorize(x, vector_size);
```

Loop Nest:

```cpp
produce input_bounded:
    for c:
        for y:
            for x:
                input_bounded(...) = ...
consume input_bounded:
    produce blur:
        for __outermost in [0, 0]<Hexagon>:
            for c:
                for y:
                    for x.x:
                        vectorized x.tmp in [0, 127]:
                            blur(...) = ...
consume blur:
```

Run on device:

Using HVX 128 schedule
Running pipeline...
Done, time: 0.0162422 s
Success!
Gaussian 5 point blur

Halide: Schedule 2 - compute_root

```cpp
vector_size = 128;
input_bounded.compute_root();
blur.compute_root().hexagon().vectorize(x, vector_size);

Loop Nest:

produce input_bounded:
  for c:
    for y:
      for x:
        input_bounded(...) = ...

consume input_bounded:
produced blur:
  for __outermost in [0, 0]<Hexagon>:
    for c:
      for y:
        for x.x:
          vectorized x.tmp in [0, 127]:
            blur(...) = ...

consume blur:

Run on device:

Using HVX 128 schedule
Running pipeline...
Done, time: 0.0162422 s
Success!
```
Gaussian 5 point blur

Halide : Schedule 3 - blur_y.compute_at

```cpp
input_bounded.compute_root();
blur_y.hexagon().compute_at(blur, y)
    .vectorize(x, vector_size, TailStrategy::RoundUp);
blur.compute_root().hexagon().vectorize(x, vector_size * 2);
```

Loop Nest:

```
produce blur:
    for __outermost in [0, 0]<Hexagon>:
        for c:
            for y:
                produce blur_y:
                    for __outermost in [0, 0]<Hexagon>:
                        for c:
                            for y:
                                for x.x:
                                    vectorized x.tmp in [0, 127]:
                                        blur_y(...) = ...

consume blur_y:
    for x.x:
        vectorized x.tmp in [0, 255]:
        blur(...) = ...
```

Run on device:

Using HVX 128 schedule
Running pipeline...
Done, time: 0.0099081 s
Success!
Gaussian 5 point blur

Halide : Best Schedule (so far).

```plaintext
input_bounded.compute_at(blur, y)
    .vectorize(x, vector_size, TailStrategy::RoundUp)
    .align_storage(x, 64)
    .store_at(blur, yo).fold_storage(y, 8);
blur_y.compute_at(blur, y)
    .vectorize(x, vector_size, TailStrategy::RoundUp);
blur.compute_root()
    .hexagon().vectorize(x, vector_size*2, TailStrategy::RoundUp)
    .split(y, yo, y, 128).parallel(yo).prefetch(y, 2);
```

Run on device:

Using HVX 128 schedule
Running pipeline...
Done, time: 0.0035454 s
Success!
Gaussian 5 Point Blur
Halide code: Schedule 4 Loop Nest

produce \texttt{blur}:
   for \_outermost in [0, 0]<Hexagon>:
      for \texttt{c}:
         parallel \texttt{y.yo}:
            store \texttt{input\_bounded}:
               for \texttt{y.y} in [0, 127]:
                  produce \texttt{input\_bounded}:
                     for \texttt{c}:
                        for \texttt{y}:
                           for \texttt{x.x}:
                              vectorized \texttt{x.tmp} in [0, 127]:
                                 \texttt{input\_bounded(...) = ...}
            consume \texttt{input\_bounded}:
            produce \texttt{blur\_y}:
               for \_outermost in [0, 0]<Hexagon>:
                  for \texttt{c}:
                     for \texttt{y}:
                        for \texttt{x.x}:
                           vectorized \texttt{x.tmp} in [0, 127]:
                              \texttt{blur\_y(...) = ...}
            consume \texttt{blur\_y}:
               for \texttt{x.x}:
                  vectorized \texttt{x.tmp} in [0, 255]:
                     \texttt{blur(...) = ...}
       consume \texttt{blur}:
Agenda

1. Halide
2. Hexagon with HVX
3. Implementation details of the Halide Compiler
4. Example 1: blur5
5. Example 2: camera_pipe
Example 2: camera_pipe

Speedup of camera_pipe on HVX (simulated) in comparison with C with intrinsics. Higher is better. C with intrinsics = 100%
Example 2: camera_pipe

Speedup of camera_pipe on HVX (simulated) in comparison with C with intrinsics. Higher is better. C with intrinsics = 100%
Example 2: camera_pipe

Speedup of camera_pipe on HVX (simulated) in comparison with C with intrinsics. Higher is better. C with intrinsics = 100%

vlut & Halide code change  Schedule change  Schedule change  Schedule change
Thank you

Follow us on:  
For more information, visit us at: 
www.qualcomm.com & www.qualcomm.com/blog

Nothing in these materials is an offer to sell any of the components or devices referenced herein.

©2016 Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.

Qualcomm is a trademark of Qualcomm Incorporated, registered in the United States and other countries. Other products and brand names may be trademarks or registered trademarks of their respective owners.

References in this presentation to “Qualcomm” may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes Qualcomm’s licensing business, QTL, and the vast majority of its patent portfolio. Qualcomm Technologies, Inc., a wholly-owned subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of Qualcomm’s engineering, research and development functions, and substantially all of its product and services businesses, including its semiconductor business, QCT.