Rapid Development of Efficient Vision Applications Using the Halide Language and CEVA Processors

CEVA®

Gary Gitelson, Yair Siegel

May 2018

Agenda

- Halide Overview
- Optimization Basics
- Creating and Optimizing Pipelines
- Halide Optimization on CEVA-XM

What is Halide?

- Domain-Specific Language for Imaging/Vision algorithms
  - Developed at MIT CSAIL
  - Separates algorithm from optimization for hardware
  - Create portable implementations
  - Optimized for different hardware (CPU, GPU and Vector DSP)

What does Halide do?

- Full compilation tool suite
- Image pipelines and optimizations “language”
- Target-oriented code compilation
  - Code generated to specific target
  - Calls native target tools (assembler/linker) to create binaries
- Specify high level optimization techniques
  - Compiler updates generated code
  - Can reach near optimal target performance

Halide Advantages

- Reuse algorithm for multiple targets
  - Develop and test on available platforms (desktop)
  - Ship to different platforms
  - Specify optimization strategy per platform
- Optimization strategies
  - Experiment quickly with various options
  - Easy per device tuning
- Easy inter-procedural optimization
  - Enables end-to-end complex pipelines
  - Achieve global best vs. local best

Who is using Halide?

- Google
  - Current Google Pixel HDR pipeline - entirely in Halide
  - Hired core MIT developers, who developed Halide
- Facebook
  - Surround 360 VR camera rig - 360 image stitching pipeline - in Halide
- Adobe
  - Publicly using Halide - specific projects unknown

Halide on CEVA-XM

- Simplify algorithm development
- Simplify target optimization, without performance compromise
  - Easier optimization experimentation
  - Inter-block optimizations easy, no major rewrites required
- Shorter feature development times for customers
- Potentially improve existing algorithms performance
- Builds optimization expertise/techniques into the tools
- Simplify memory management, system and host interaction

Agenda

- Halide Overview
- Optimization Basics
- Creating and Optimizing Pipelines
- Halide Optimization on CEVA-XM

Optimizations Types

- Variety of optimization strategies:
  - Splitting
  - Re-ordering
  - Tiling
  - Vectorization
  - Loop Unrolling
  - Parallelization

Details in upcoming slides

Tiling

- Processing in tiles improves cache locality

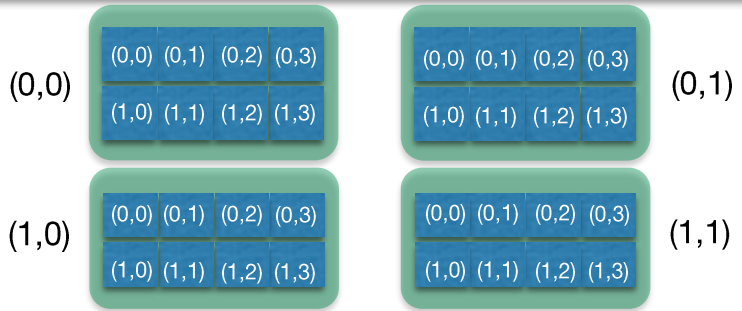
Before tiling, processing in simple scanline order:

```
for (y = 0; y < height; y++) {
    for (x = 0; x < width; x++) {
        out[y][x] = in[y][x] * 1.1;
    }
}
```

After tiling, introduce loops to process in smaller tiles, using scanline order within each tile:

```
for (yo = 0; yo < height; yo = yo + 2) {
    for (xo = 0; xo < width; xo = xo + 4) {
        for (yi = 0; yi < 2; yi++) {
            for (xi = 0; xi < 4; xi++) {
                out[yo + yi][xo + xi] = in[yo + yi][xo + xi] * 1.1;
            }
        }
    }
}
```

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)



Vectorizing

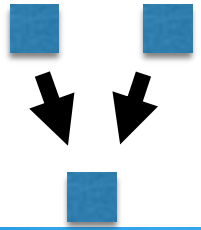
- Advanced processors such as CEVA-XM, have special instructions to process multiple data elements in parallel

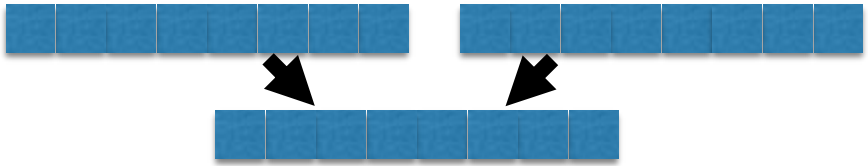
Before vectorization -
Output produced - one element each time:

```
for (y = 0; y < height; y++) {
    for (x = 0; x < width; x++) {
        out[y][x] = in_a[y][x] + input_b[y][x];
    }
}
```

After vectorization -
Entire vector accessed and generated in a single step:

```
for (y = 0; y < height; y=y+8) {    for (x = 0; x < width; x=x+8) {
      //< Vec-C instructions to load 8 values >
      //< Vec-C instructions for addition >
      //< Vec-C instructions to store 8 values >
   }
}
```



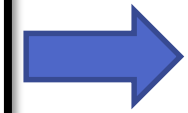


Loop Unrolling

- Often better to process more than single element in loops
- Potentially re-use loaded data to reduce loads and stores
- Better utilizes processor execution pipeline
- Reduces branch decision overhead

Without Unroll - Single pixel processed per loop iteration:

```
for (y = 0; y < height; y++) {
    for (x = 0; x < width; x++) {
        out[y][x] = in[y][x] * 1.1;
    }
}
```



With Unrolling - 4 pixels processed per iteration:

```
for (y = 0; y < height; y = y++) {
    for (x = 0; x < width; x = x + 4) {
        out[y][x] = in[y][x] * 1.1;
        out[y][x + 1] = in[y][x + 1] * 1.1;
        out[y][x + 2] = in[y][x + 2] * 1.1;
        out[y][x + 3] = in[y][x + 3] * 1.1;
    }
}
```

Algorithm vs. Optimization

- In Halide, specify algorithm once, in mathematical form
- Expressions describe output as operation on input

```
brighten(y, x) = input(y,x) * 1.1;
```

- Optimizations specified separately
- Optimizations applied without code changes in algorithm
- Powerful way to experiment with optimization
- Easy to combine optimizations cross algorithmic boundaries

```
brighten.vectorize(x,8).unroll(x,2).parallel(y,4);
```

Agenda

- Halide Overview
- Optimization Basics
- Creating and Optimizing Pipelines
- Halide Optimization on CEVA-XM

How to Build a Pipeline

- Halide defines functions from input to output
- Functions can be chained
- Chaining creates pipelines

```
black_level(x,y) = input(x,y) - 10;
brighten(x,y) = black_level(x,y) * 1.1;
blur_x(x,y) = (brighten(x-2,y) + brighten(x-1,y) + brighten(x,y) +
      brighten(x+1,y) + brighten(x+2, y)) / 5;
blur_y(x,y) = (blur_x(x,y-2) + blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1) +
       blur_x(x,y+2)) / 5;
```

- Pipelines can be optimized as well

Pipeline Optimization (1/3)

- Two connected pipeline stages will fold producer code into consumer
- Example:

```
black_level(x,y) = input(x,y) - 10;
brighten(x,y) = black_level(x,y) * 1.1;
output(x,y) = brighten(x,y);
```

- Equivalent to:

```
output(x,y) = (input(x,y) – 10) * 1.1;
```

- Operations folded together - Multiple algorithms merge into single block
- Good in some cases, inefficient in others

Pipeline Optimization (2/3)

compute_root, store_root

- Optional attributes - each pipeline stage runs independently

```
black_level.compute_root();
brighten.compute_root();
```

- Each stage creates and stores output before next stage runs

Pipeline Optimization (3/3)

compute_at, store_at

- Optional attributes - pipeline stages incorporated to their consumer
- Forms tightly coupled producer/consumer relationships

```
black_level.compute_at(brighten, y); // produce black_level, each line of brighten
brighten.compute_root();
```

- Each stage produces/stores data required for specified consumer
- Pipelines specify to produce at (as example): tile / line / pixel level

Putting it all Together

- Write core algorithms in Halide
  - Implement a series of pipeline stages
  - Link them together into a pipeline

```
black_level(x,y) = input(x,y) - 10;
brighten(x,y) = black_level(x,y) * 1.1;
blur_x(x,y) = (brighten(x-2,y) + brighten(x-1,y) + brighten(x,y) +
      brighten(x+1,y) + brighten(x+2, y)) / 5;
blur_y(x,y) = (blur_x(x,y-2) + blur_x(x,y-1) + blur_x(x,y) + blur_x(x,y+1) +
      blur_x(x,y+2)) / 5;
```

- Test and validate correctness on desktop (ref platform)
- Same code (different optimizations) shipped on multiple platforms

Agenda

- Halide Overview
- Optimization Basics
- Creating and Optimizing Pipelines
- Halide Optimization on CEVA-XM

Halide on CEVA Platforms

- Halide supports many platforms
- CEVA-XM Halide compiler specifics
  - Maps Halide constructs to CEVA instruction set
  - Additional optimization passes for specific CEVA-XM architecture
  - DMA constructs extending Halide language
  - CEVA-specific runtime for memory management and kernel invocation
  - Integration of Halide compiler with CEVA-XM tool suite
- Enable Halide programs to generate native CEVA binaries directly for execution on CEVA-XM4 and CEVA-XM6 Vision DSPs

CEVA Specific Support

- Halide scheduling describes optimizations and mappings to hardware
- Introduced new keywords to indicate code generation for CEVA-XM
- CEVA-XM targeting requires minimal changes
  - ***"ceva()"*** keyword - generates code for CEVA-XM target

    Example: brighten.ceva().compute_root();
  - A small block of code indicates the platform has CEVA-XM4 processor
- DMA control
  - ***"dma_in()"*** and ***"dma_out()" –*** added scheduling keywords
  - Indicate (loop level) granularity DMA used for particular input or output buffer
  - Additional constructs indicate DMA transaction size (number of lines)

DMA Scheduling

- Introducing DMA processing adds additional scheduling

  Example:

```
// Create a function to wrap DMA transaction
// DMA one line per time, reading in 4 lines ahead of the current processing
input_func.in().compute_at(blur_y, y).store_root().fold_storage(y, 4).dma_in(y);

// process input through “blur_y” algorithm, schedule for CEVA DSP, vectorize 32 pixels
// buffer 2 lines before initiating DMA
blur_y.ceva().vectorize(x, 32).compute_at(output, y).store_root().fold_storage(y, 2);

// DMA output one line at a time
output.compute_root().ceva().dma_out(y);
```

- Generated code includes all necessary DMA transactions, synchronizing, buffer allocation

Putting it all Together

- Schedule on CEVA-XM
  - ”ceva()” keyword - tools generate code targeting CEVA DSP

```
black_level.ceva().vectorize(x,32).unroll(x).compute_at(brighten,y);
brighten.ceva().vectorize(x,16).unroll(x).compute_root();
blur_x.ceva().vectorize(x,32).compute_at(blur_y, x);
blur_y.ceva().vectorize(x, 32).compute_root();
```

  - Generates all related runtime system code from host/CPU
    - Both host/CPU code and DSP code generated as single unit
    - Handles buffer management, data transfers, kernel launches and other system functions
  - DMA usage for inputs and output management possible
  - Experiment with various optimization strategies to achieve best performance
  - Ship!

Halide Performance

- Simple benchmark performance
  - 1.5-3x performance degradation seen vs. optimal CEVA Vec-C
    - 1-2x near-term target
- Real world performance
  - As algorithm complexity increases, writing optimal low-level code is increasingly difficult
  - As pipelines grow, inter-stage optimization becomes a necessity
  - Halide shines -
    - Rapid experimentation with wide range of optimization strategies
    - Extremely efficient and easy to optimize inter-stage pipelines

Halide enables near optimal performance,
while greatly improving developer productivity and code quality

Resources

- Halide Website - http://halide-lang.org/
- Developer mailing list archive – https://lists.csail.mit.edu/mailman/listinfo/halide-dev/
- Halide SIGGRAPH paper - http://people.csail.mit.edu/jrk/halide12/
- Halide PLDI paper - http://people.csail.mit.edu/jrk/halide-pldi13.pdf
- Halide GitHub Repository - https://github.com/halide/Halide
- CEVA Vision DSPs - https://www.ceva-dsp.com/app/imaging-computer-vision/

Thank You!

Yair Siegel: Yairs@ceva-dsp.com

Gary Gitelson: gary@mperpetuo.com

www.CEVA-DSP.com