Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines, *PLDI 2013*

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris,

Frédo Durand, Saman Amarasinghe

Presented by: Zineb Benameur El Youbi,
Sanjay Sri Vallabh Singapuram
and Hannah Potter

Outline

- Motivation:
  - Why Halide?
  - What is Halide?
  - The Halide DSL
- Implementation:
  - Scheduling Image Processing Pipelines
  - Compiling Scheduled Pipelines
  - Autotuning Pipeline Schedules
- Results
- Analysis

Motivation

We are surrounded by computational cameras

Image processing pipelines are everywhere!

- Capturing, analyzing , mining, rendering visual information
- Applications : Instagram, Adobe, etc.

$\Rightarrow$ Demand extremely high performance to cope with high rising resolution, frame rate, and complexity of algorithms

Motivation

Example : 3x3 blur

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

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

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

Hand optimized C++ , **x11 faster**

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

Writing fast image processing pipelines is **hard**

Optimization => Transform program & Data Structure

Halide's **answer**?

Separate Algorithm from Schedule aka Execution Strategy

Algorithm : What is computed
Schedule: Where and **When** it's computed

Easy for programmers to build pipelines
    Simplifies algorithm code
    Improves modularity

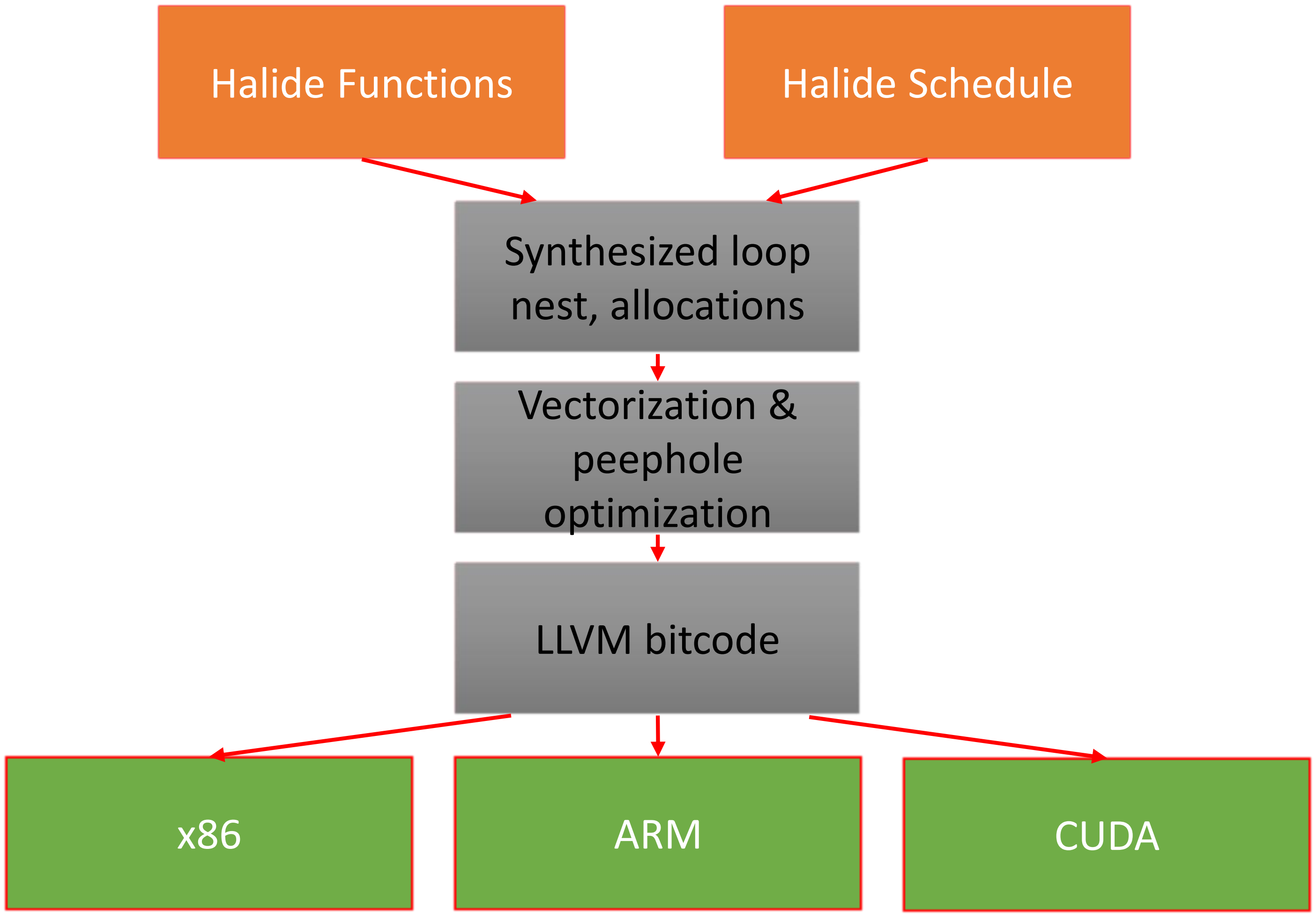
Easy for programmers to specify and explore optimizations
    Fusion, tiling, parallelism, vectorization
    And **NOT BREAK THE ALGORITHM**

Easy for the compiler to generate code

What is **Halide**?

- A Domain Specific Language (DSL)
- Write high performance code easily
- Front end embedded in C++
- Compiler targets: x86/SSE, ARM v7/NEON, CUDA, Native Client, OpenCL, and Metal

What is **Halide**?



Halide **DSL**

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

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

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

Describe image processing pipelines in a simple functional style

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

Pipeline stages are **functions** from coordinates to value

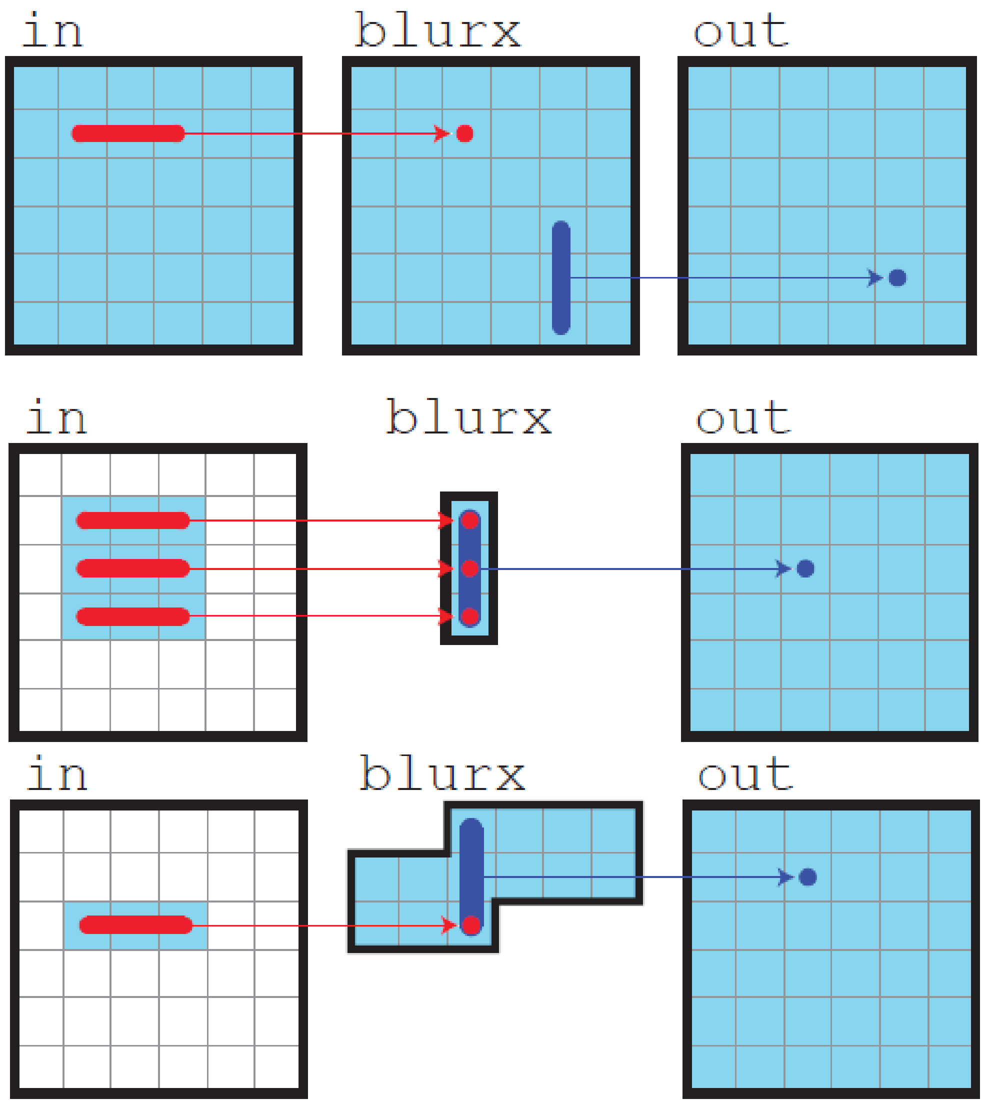
Execution order and storage are unspecified

Outline

- Motivation:
  - Why Halide?
  - What is Halide?
  - The Halide DSL
- Implementation:
  - Scheduling Image Processing Pipelines
  - Compiling Scheduled Pipelines
  - Autotuning Pipeline Schedules
- Results
- Analysis

Schedule Space

- Breadth-first
- Depth-first (loop-fusion)
- Sliding-window



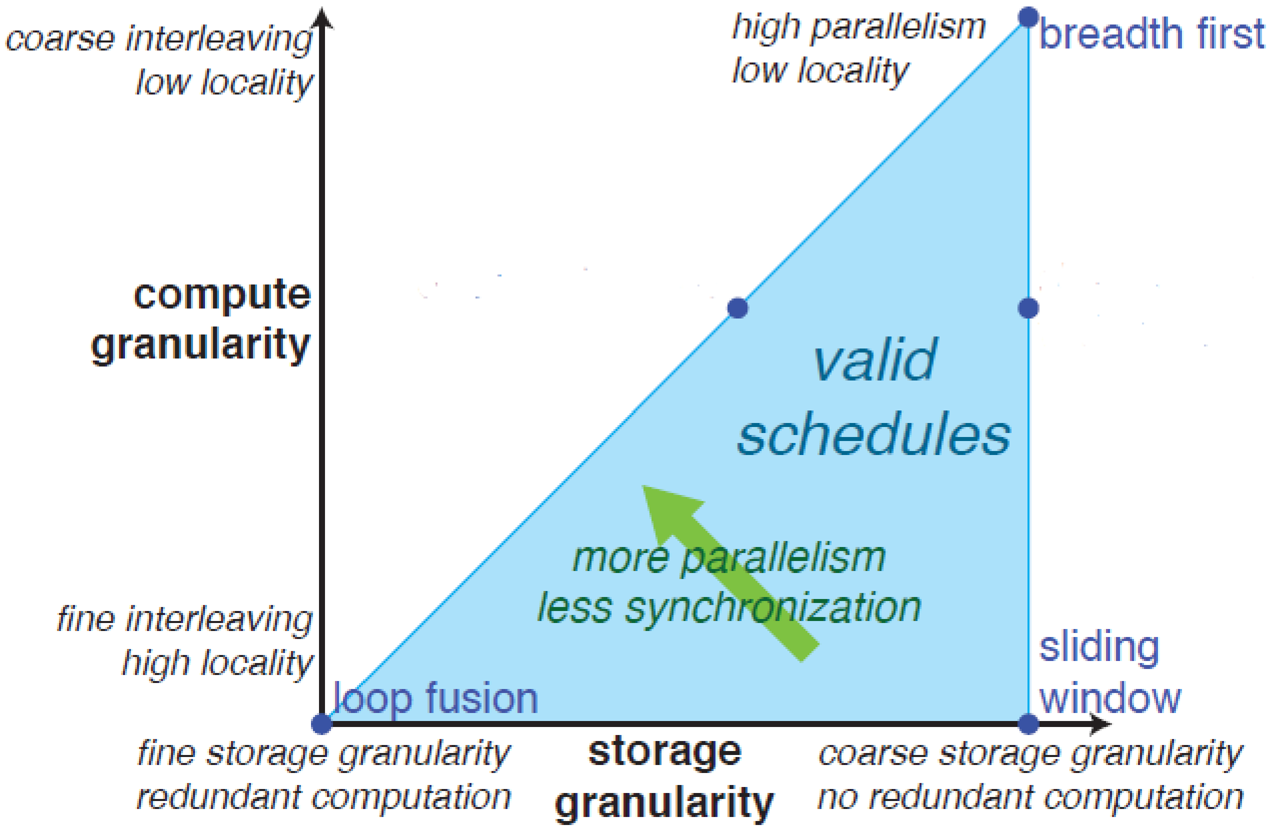
Defining a schedule

- Domain Order: Specifies order of nested iterations

```
order(y, x) =>
for y in Y_MIN … Y_MAX:
  for x in X_MIN … X_MAX:
```

- Call Schedule:
  - Compute granularity: Where to compute?
  - Storage granularity: How long to store?

Compute and Storage Granularity



Compute granularity <= Storage granularity
Can't compute more than available storage !!

Domain Order

- Reorder: `order(y, x)` to `order(x, y)`
- Tiling: `split(x, 8) -> order(tx, x)`
- Vectorize: `order(tx, y, x).vectorize(x)`
- Parallelize: `order(tx, y, x).parallel(tx)`
- Strict order: `order(tx, y, x).sequential(y)`

Call Schedule

Sliding-window: blurx: `store @ out.`x_0 , `compute @ out.`y_1

```
par for out.y0 in 0 ... out.y.extent/4
 for out.x0 in 0 ... out.x.extent/4
  alloc blurx[blurx.x.extent][blurx.y.extent]
  for out.y1 in 0 ... 4
   // compute blurx
   vec for out.x1 in 0 ... 4
     // compute out(4*x0 + x1, 4*y0 + y1)
```

Code-generation

1. Representation in Halide-IR
2. Optimizations
   1. Storage Folding (reduces storage granularity)
   2. Sliding-Window Detection (increases storage granularity)
3. Back-end Code-gen
   - Lower to LLVM-IR
   - GPU Code-gen
     - Loop extents must respect thread and block limitations
     - Handle data movement between host and GPU
     - Automating code-gen greatly improves programmer productivity!

Auto-tuning

- State-space explosion!
  - `len(states(LaplacianFilters)) > 10`^{720}
- Stochastic state-space exploration
  - Start with reasonable initial state.
  - Mutate schedule to see if mutation is better
  - Petabricks Autotuner
- Technique applied to other domains
  - ASTRA: Exploiting Predictability to Optimize Deep Learning [ASPLOS ’19]

Outline

- Motivation:
  - Why Halide?
  - What is Halide?
  - The Halide DSL
- Implementation:
  - Scheduling Image Processing Pipelines
  - Compiling Scheduled Pipelines
  - Autotuning Pipeline Schedules
- Results
- Analysis

Results

- Examples use variety of algorithms and communication patterns
- Pipelines have 2-99 stages

	# functions	# stencils	graph structure
Blur	2	2	simple
Bilateral grid	7	3	moderate
Camera pipeline	32	22	complex
Local Laplacian filters	99	85	very complex
Multi-scale interpolation	49	47	complex

Results

x86

	Halide tuned (ms)	Expert tuned (ms)	Speedup	Lines Halide	Lines expert	Factor shorter
Blur	11	13	1.2×	2	35	18×
Bilateral grid	36	158	4.4×	34	122	4×
Camera pipe	14	49	3.4×	123	306	2×
Interpolate	32	54	1.7×	21	152	7×
Local Laplacian	113	189	1.7×	52	262	5×

CUDA

	Halide tuned (ms)	Expert tuned (ms)	Speedup	Lines Halide	Lines expert	Factor shorter
Bilateral grid	8.1	18	2.3×	34	370	11×
Interpolate	9.1	*54**	*5.9*×	21	*152**	7×
Local Laplacian	21	*189**	*9*×	52	*262**	5×

Analysis

Strengths

- Reduced developer time
  - Local Laplacian Filters
    - 2-3 weeks for expert to hand-optimize
    - 1 day to write in Halide
- Shorter & less complex programs
- Autotuning is target specific
  - Take advantage of specific architectures (e.g. CPU vs. GPU)
- Faster programs

Weaknesses/Limitations

- Limited to rectangular image processing
- Compilation time
  - 2 hours – 2 days to run autotuning
- Autotuning is target specific
  - Schedules may work poorly on different architectures
- Tuner can get stuck in local minima
  - Requires restart with new random initialization